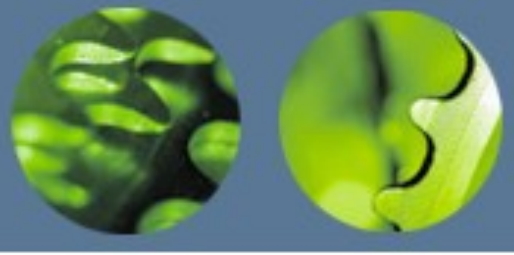
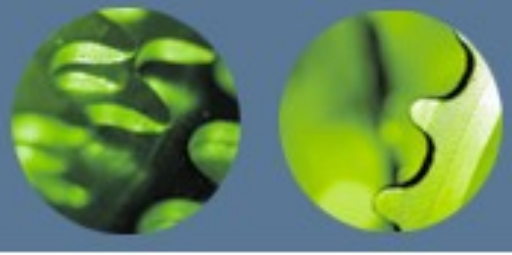


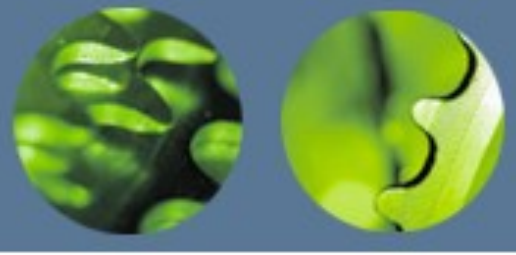
# Solving Problems by Searching

- ✓ Searching the process finding the required states or nodes.
- ✓ Searching is to be performed through the state space.
- ✓ Search process is carried out by constructing a search tree.
- ✓ Search is a universal problem-solving technique.



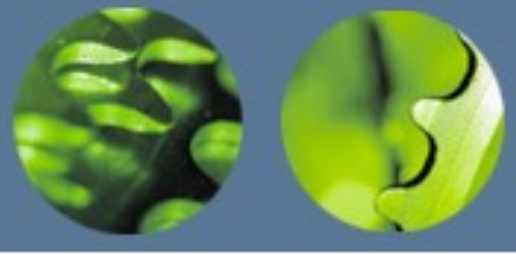
# Solving Problems by Searching

- ✓ Search involves systematic trial and error exploration of alternative solutions.
- ✓ Useful when the sequence of actions required to solve a problem is not known
  - Path finding problems, e.g, eight puzzle, traveling salesman problem
  - Two player games, e.g., chess and tic tac toe
  - Constraint satisfaction problems, e.g., eight queens



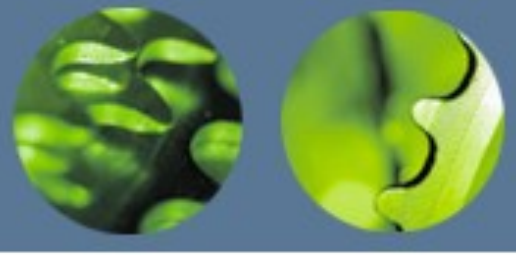
# Evaluating Search Strategies

- ✓ Completeness
  - – Guarantees finding a solution whenever one exists
- ✓ Time complexity
  - – How long (worst or average case) does it take to find a solution?
  - – Usually measured in terms of the number of nodes expanded
- ✓ Space complexity
  - – How much space is used by the algorithm?
  - – Usually measured in terms of the maximum size of the “nodes” list during the search
- ✓ Optimality/Admissibility
  - – If a solution is found, is it guaranteed to be an optimal one?
  - –That is, is it the one with minimum cost?



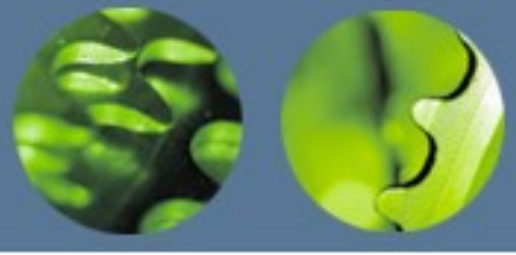
# Uninformed vs. Informed Search

- ✓ Uninformed search strategies
  - – Also known as "blind search," uninformed search strategies use no information about the likely "direction" of the goal node(s)
  - – Uninformed search methods: Breadth-first, depth-first, depth-limited, uniform-cost, depth-first iterative deepening, bidirectional
- ✓ Informed search strategies
  - – Also known as "heuristic search," informed search strategies use information about the domain to (try to) (usually) head in the general direction of the goal node(s)
  - – Informed search methods: Hill climbing, best-first, greedy search, A*, Simulated Annealing.



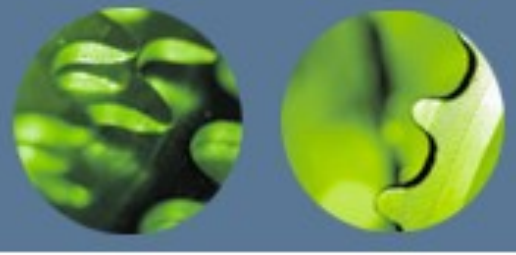
# Uninformed Search

- ✓ These types of search strategies are provided with the problem definition and these don't have additional information about the state space.
- ✓ These can only expand current state to get a new set of states and distinguish a goal state from non-goal state.
- ✓ Uninformed search strategies use only the information available in the problem definition.
- ✓ Less effective than informed search.



# Uninformed Search

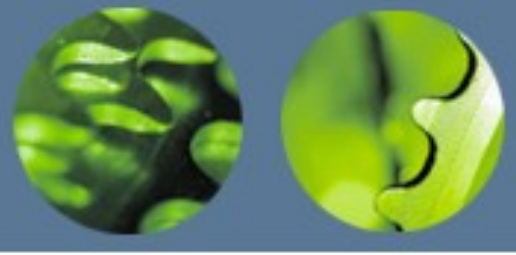
- ✓ We call search algorithms that do not use any extra information regarding the problem or our representation of the problem, "blind".
- ✓ These searches typically visit all of the nodes of a tree in a certain order looking for a per-specified goal.
  - No cleverness is used to decide where to look next.
  - Searches may never find the goal state.
  - In some cases blind search is the right approach.



# Uninformed Search

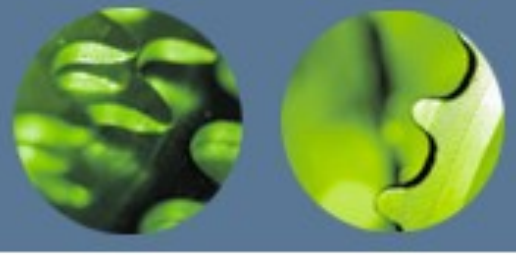
- ✓ Types

1) Breadth-first search
2) Uniform Cost Search
3) Depth-first search
4) Depth-limited search
5) Iterative Deepening Depth-first search
6) Bidirectional Search



# Breadth-First Search (BFS)

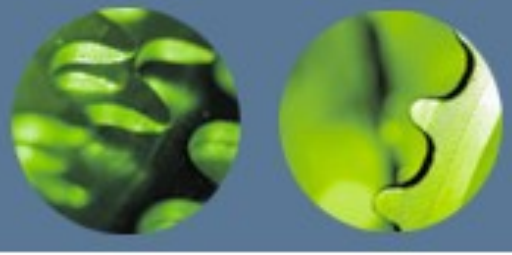
- ✓ Proceeds level by level down the search tree
- ✓ Starting from the root node (initial state) explores all children of the root node, left to right
- ✓ If no solution is found, expands the first (leftmost) child of the root node, then expand the second node at depth 1 and so on …



# Breadth-First Search (BFS)

- ✓ Process

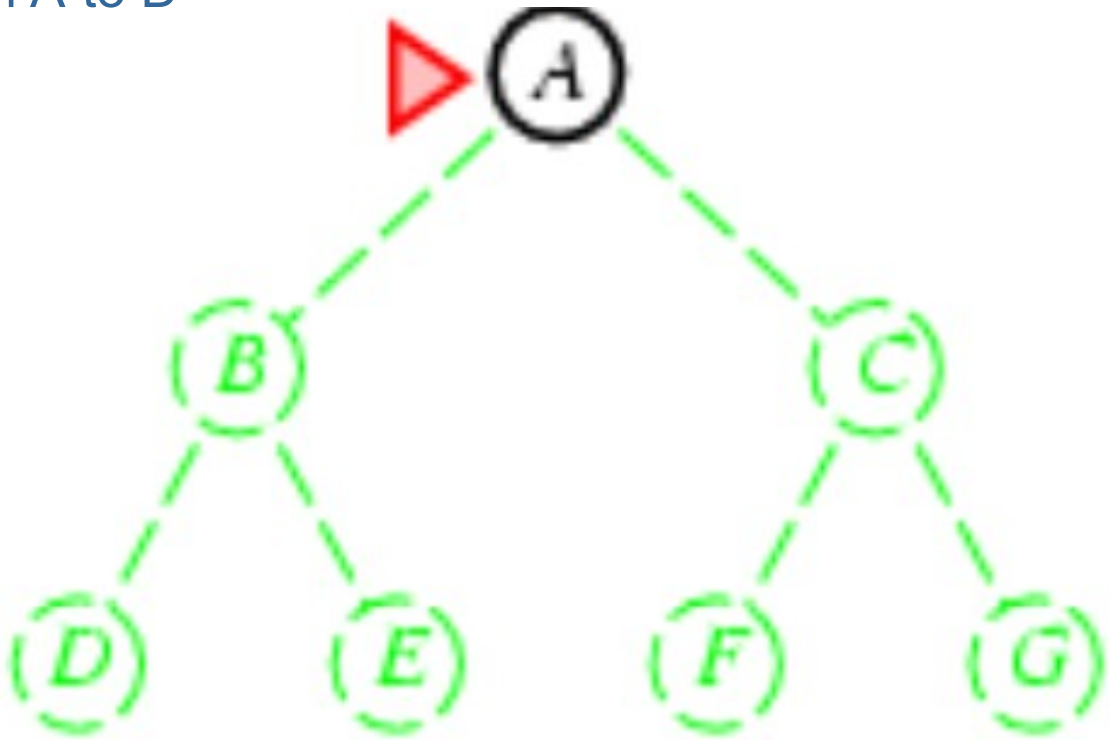
  - Place the start node in the queue(FIFO Data Structure)
  - Do while the queue is not empty
    - Dequeue a node
    - If the node is the goal then Display “Success” and stop.
    - Else, enqueue all the unvisited childrens of the node into the queue.

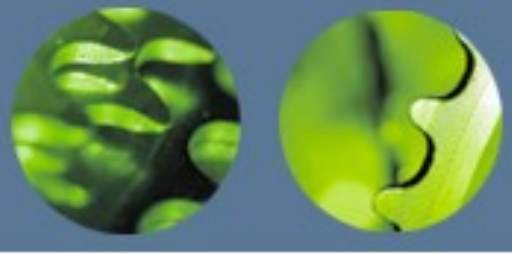


# Breadth-First Search (BFS)

Find path from A to D

Is A a goal state?

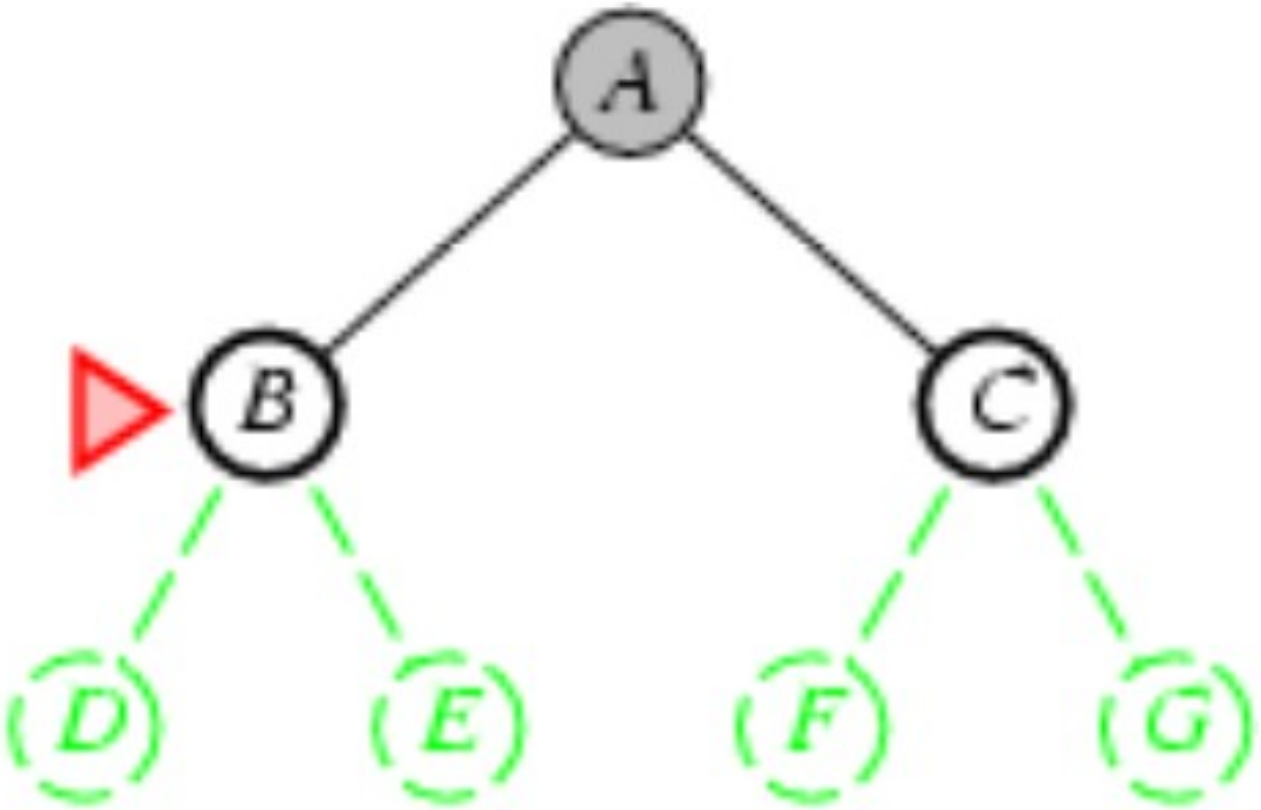




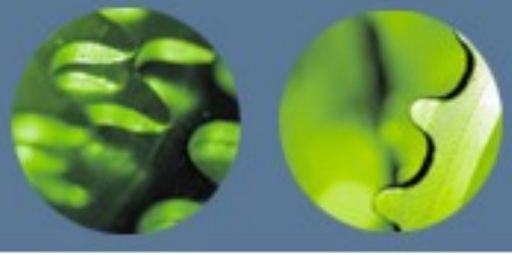
# Breadth-First Search (BFS)

Expand:
fringe = [B,C]

Is B a goal state?



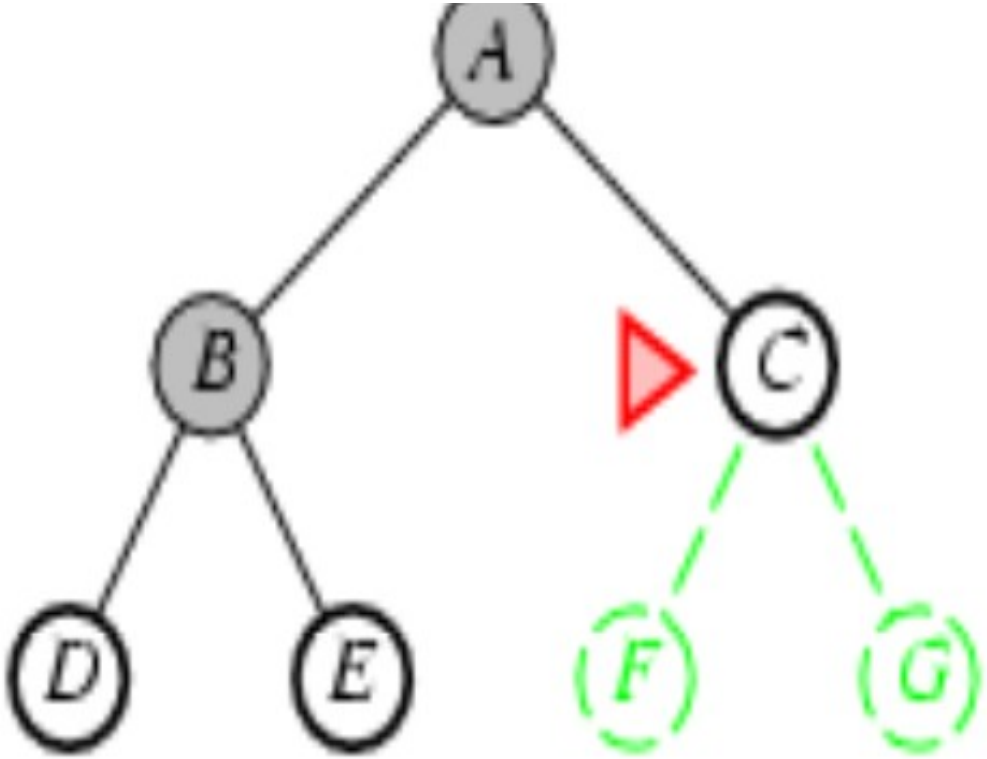

fringe is a FIFO queue, i.e., new successors go at end

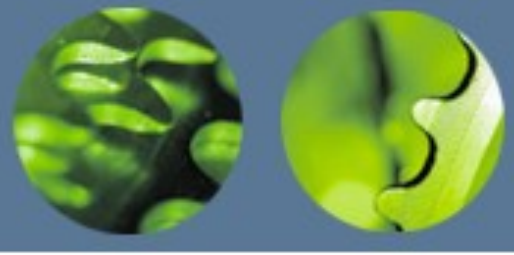


# Breadth-First Search (BFS)

Expand:
fringe=[C,D,E]

Is C a goal state?

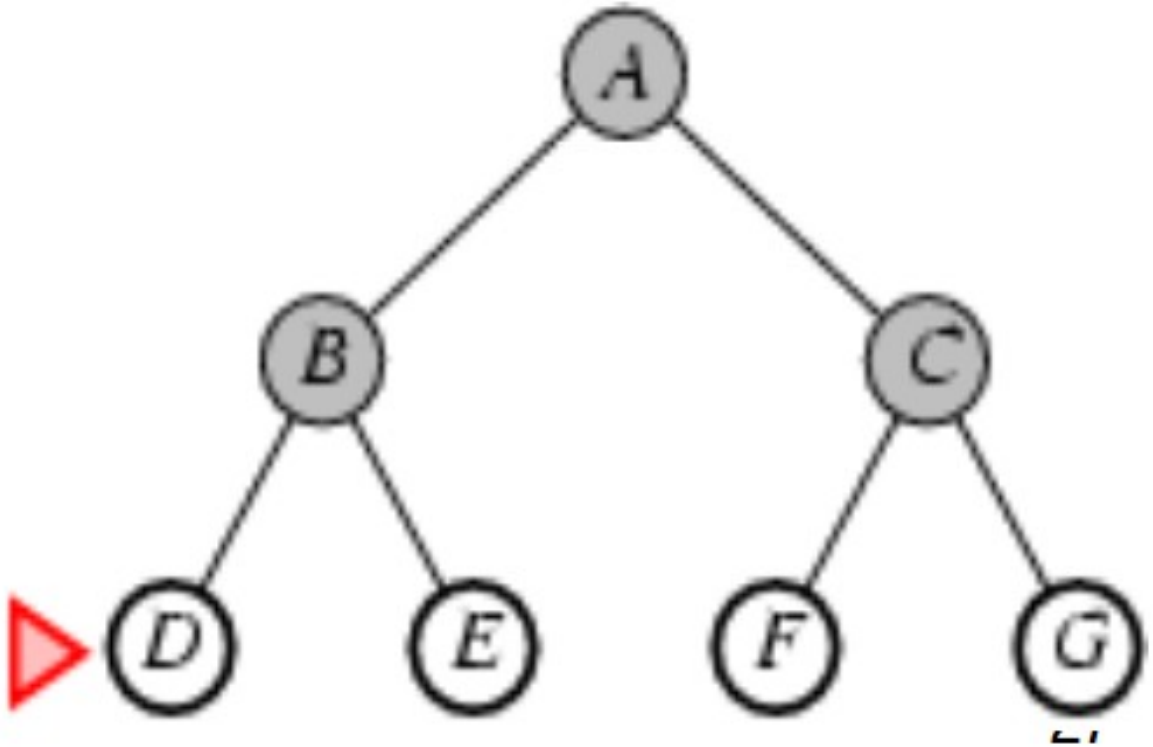




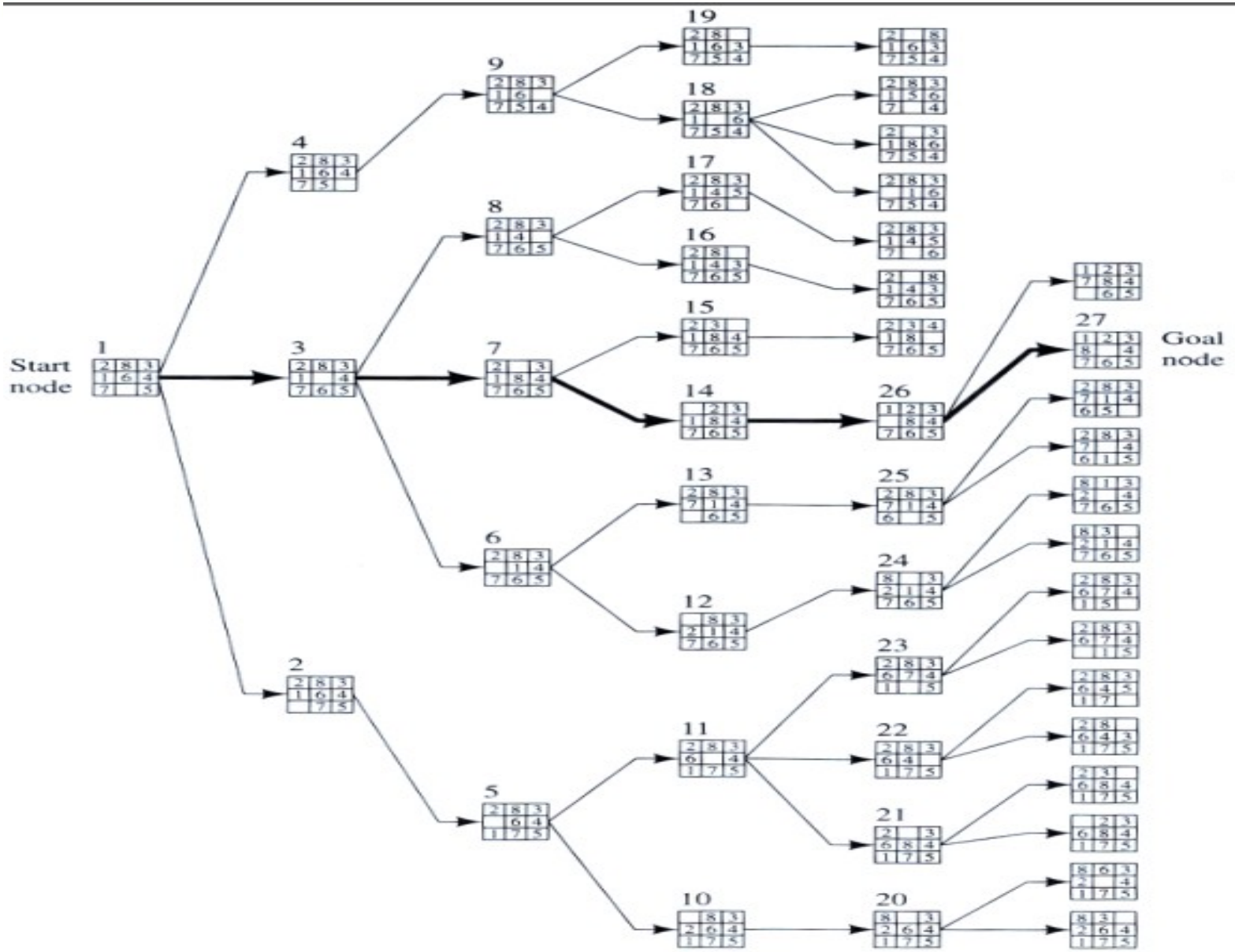
# Breadth-First Search (BFS)

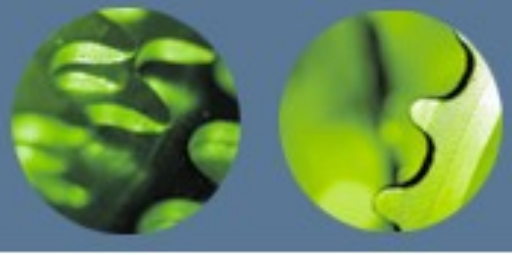
Expand:
fringe=[D,E,F,G]

Is D a goal state?



# Breadth-First Search (BFS)





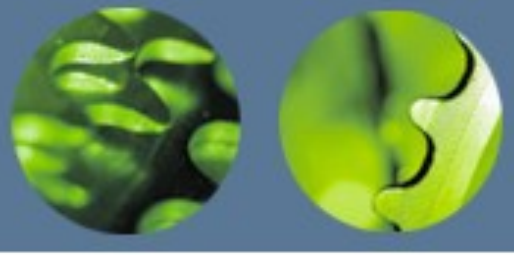
# Breadth-First Search (BFS)

## Properties of Breadth-First Search

- ***Completeness:*** Complete if the goal node is at finite depth
- ***Optimality:*** It is guaranteed to find the shortest path
- ***Time complexity***
  - For branching factor b and depth level d
  - Expand root yields b nodes at $1^{st}$ level
  - Expand $1^{st}$ level yields $b^2$ noes at $2^{nd}$ level.
  - If the goal is in $d^{th}$ level, in the worst case, the goal node would be the last node in the $d^{th}$ level.

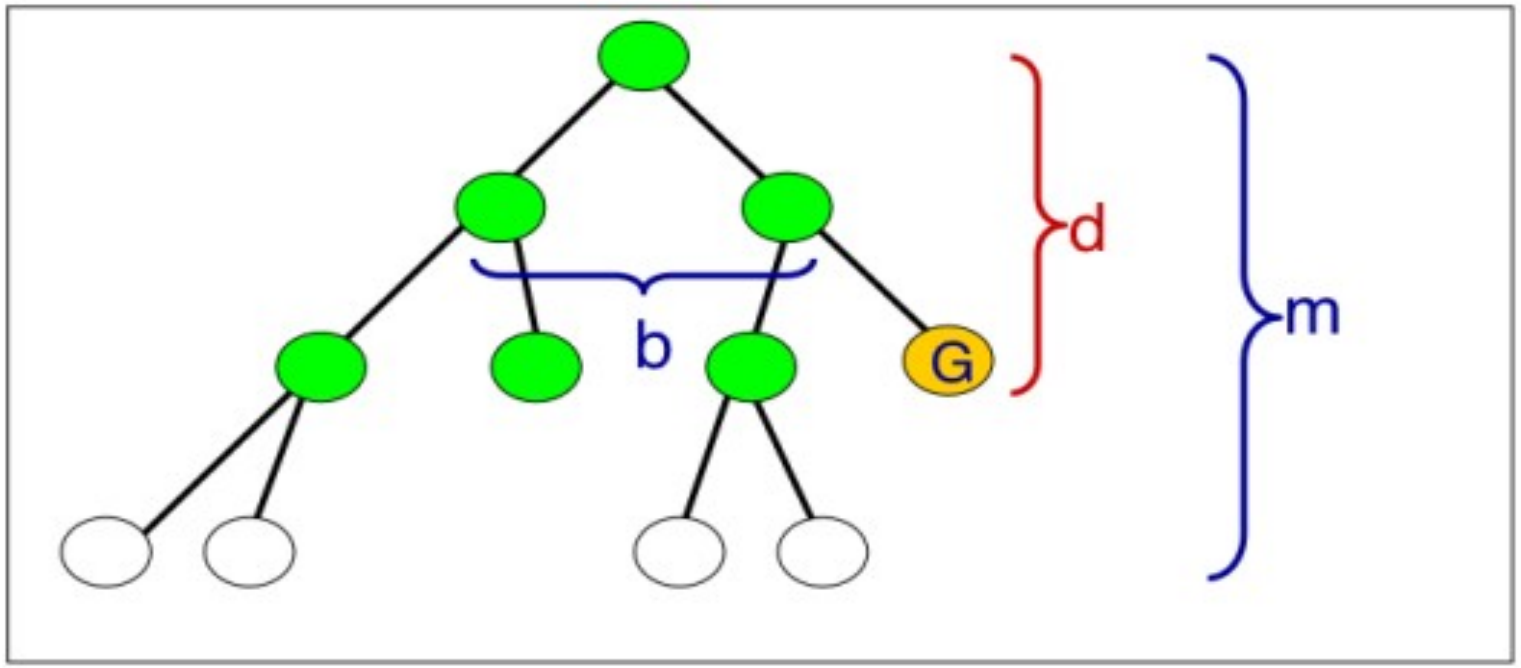
  - We should expand $(b^d-1)$ nodes in the $d^{th}$ level (except the goal node itself). Total nodes in $d^{th}$ level = $b(b^d-1)= b^{d+1}-b$
  - Total no of nodes generated = $1+b+b^2+b^3+\ldots..+b^{d+1}-b$
  - Time complexity = $O(b^{d+1})$

- ***Space Complexity***: $O(b^{d+1})$

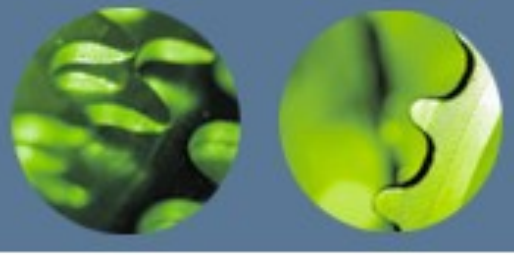


# Time complexity of BFS

- If a goal node is found on depth d of the tree, all nodes up till that depth are created and examined (note: and the children of nodes at depth d are created and enqueued, but not yet examined).

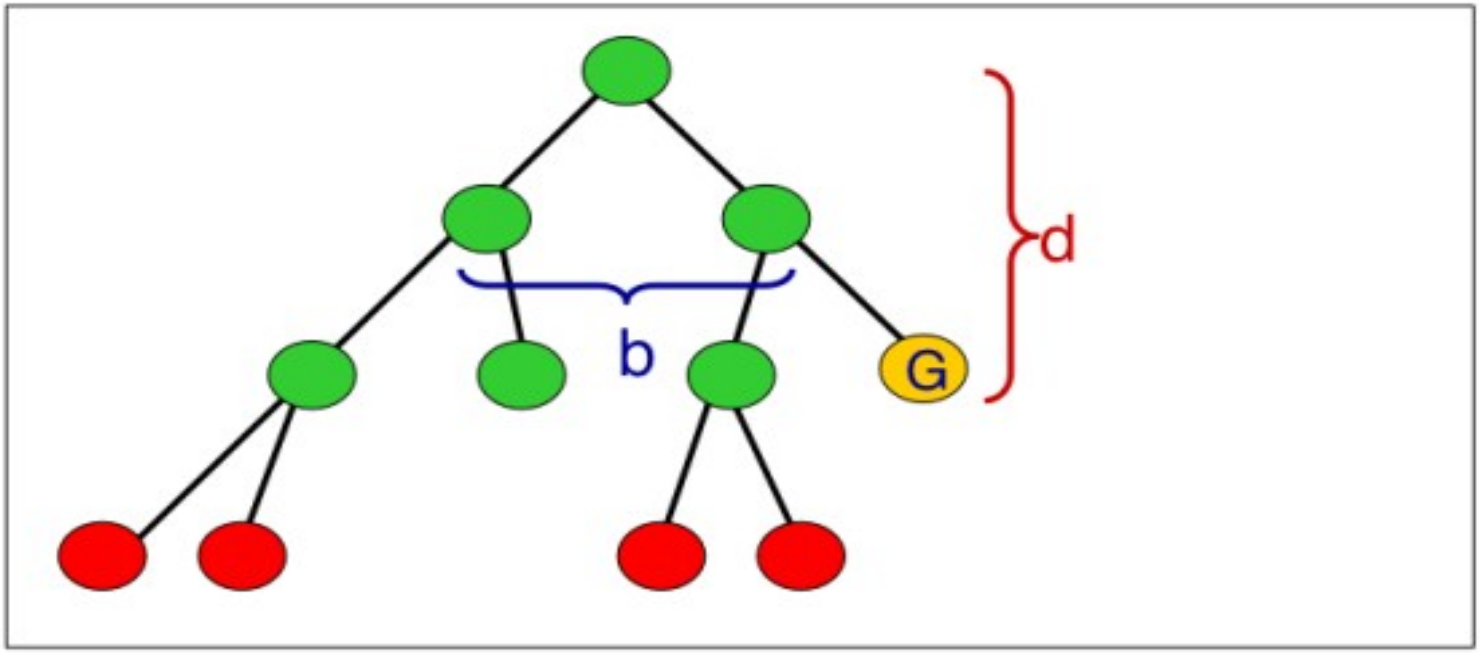



- Thus: $O(b^d)$

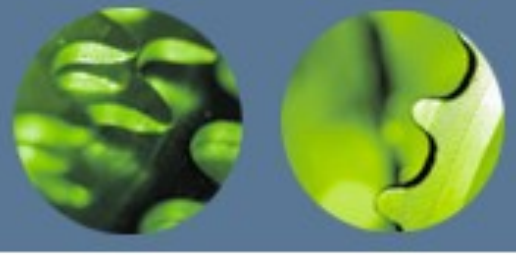


# Space complexity of BFS

- Largest number of nodes in QUEUE is reached on the level d+1 just beyond the goal node.




- QUEUE contains all  nodes. (Thus: 4) .
- In General: $b^{d+1} - b \sim b^d$

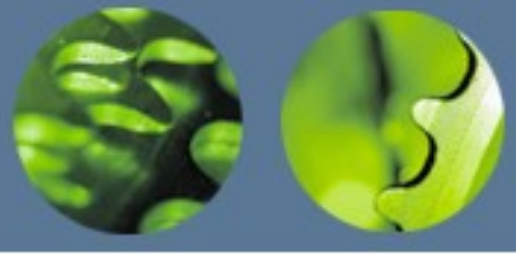


# BFS Weakness

- Exponential Growth

***Time and memory requirements for breadth-first search, assuming a branching factor of 10, 100 bytes per node and searching 1000 nodes/second***

| Depth | Nodes | Time | | Memory | |
|---|---|---|---|---|---|
| 0 | 1 | 1 | millisecond | 100 | kbytes |
| 2 | 111 | 0.1 | second | 11 | kilobytes |
| 4 | 11,111 | 11 | seconds | 1 | megabyte |
| 6 | $10^6$ | 18 | minutes | 111 | megabytes |
| 8 | $10^8$ | 31 | hours | 11 | gigabytes |
| 10 | $10^{10}$ | 128 | days | 1 | terabyte |
| 12 | $10^{12}$ | 35 | years | 111 | terabytes |
| 14 | $10^{14}$ | 3500 | years | 11,111 | terabytes |



# Uniform Cost Search

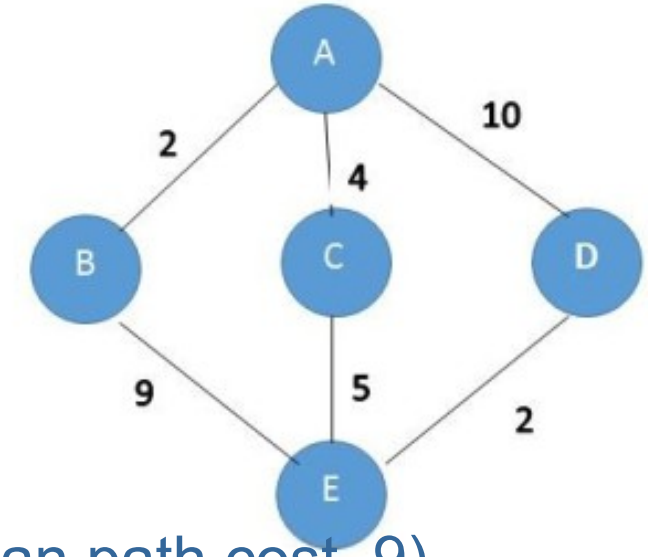
- ✓ Breadth first Search finds the shallowest goal but it's not always sure to find the optimal solution.
- ✓ Uniform cost search can be used if the cost of traveling from one node to another is available.
- ✓ Uniform cost search always expands the lowest cost node on the fringe (the collection of nodes that are waiting to be expanded.)
- ✓ The first solution is guaranteed to be the cheapest one because a cheaper one is expanded earlier and so would have been found first

# Uniform Cost Search

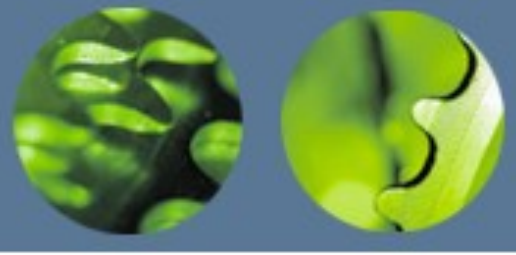
Find path from A to E

- ✓ Expand A to [B,C,D]
- ✓ The path to B is the cheapest one with path cost 2.
- ✓ Expand B to E
  - – Total path cost = 2+9 =11
- ✓ This might not be the optimal solution

since the path AC has path cost 4 (< 11 )

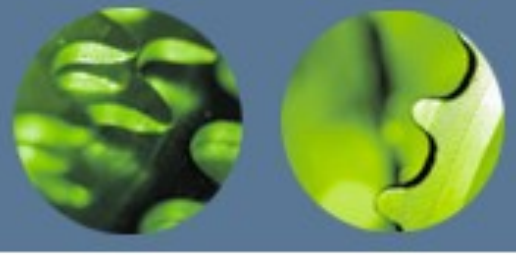



- ✓ Expand C to E
  - – Total path cost = 4+5 =9
- ✓ Path cost from A to D is 10 ( greater than path cost, 9)
- ✓ Hence optimal path is ACE



# Uniform Cost Search

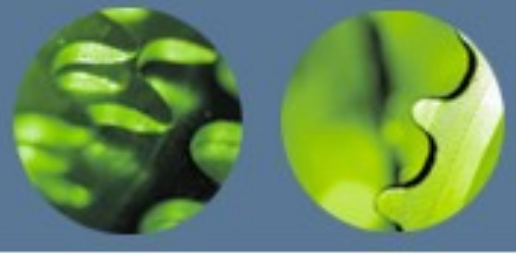
- ✓ The thing to remember is it does not care about the no of steps a path has but only about their cost.
- ✓ Hence it might get stuck in an infinite loop if it expands a node that has a zero cost action leading back to same state



# Uniform Cost Search

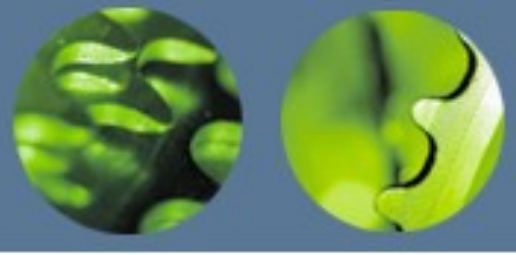
**Properties of uniform cost search**

- ***Completeness*:** Complete if the cost of every step is greater than or equal to some small positive constant **e**
- ***Optimality*:** Optimal if the cost of every step is greater than or equal to some small positive constant **e**
- ***Time complexity* :** $O(b^{c^*/e})$ where c* is cost of optimal path, e is small positive constant
- ***Space Complexity* :** $O(b^{c^*/e})$



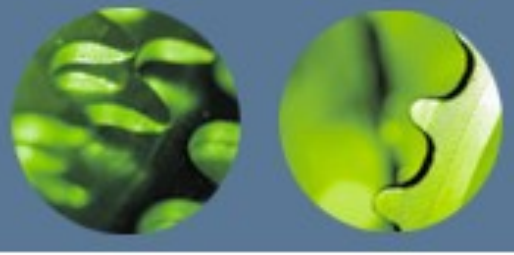
# Depth-First Search (DFS)

- ✓ DFS proceeds down a single branch of the tree at a time.
- ✓ It expands the root node, then the leftmost child of the root node, then the leftmost child of that node etc.
- ✓ Always expands a node at the deepest level of the tree
- ✓ Only when the search hits a dead end (a partial solution which can't be extended) does the search backtrack and expand nodes at higher levels.



# Depth-First Search (DFS)

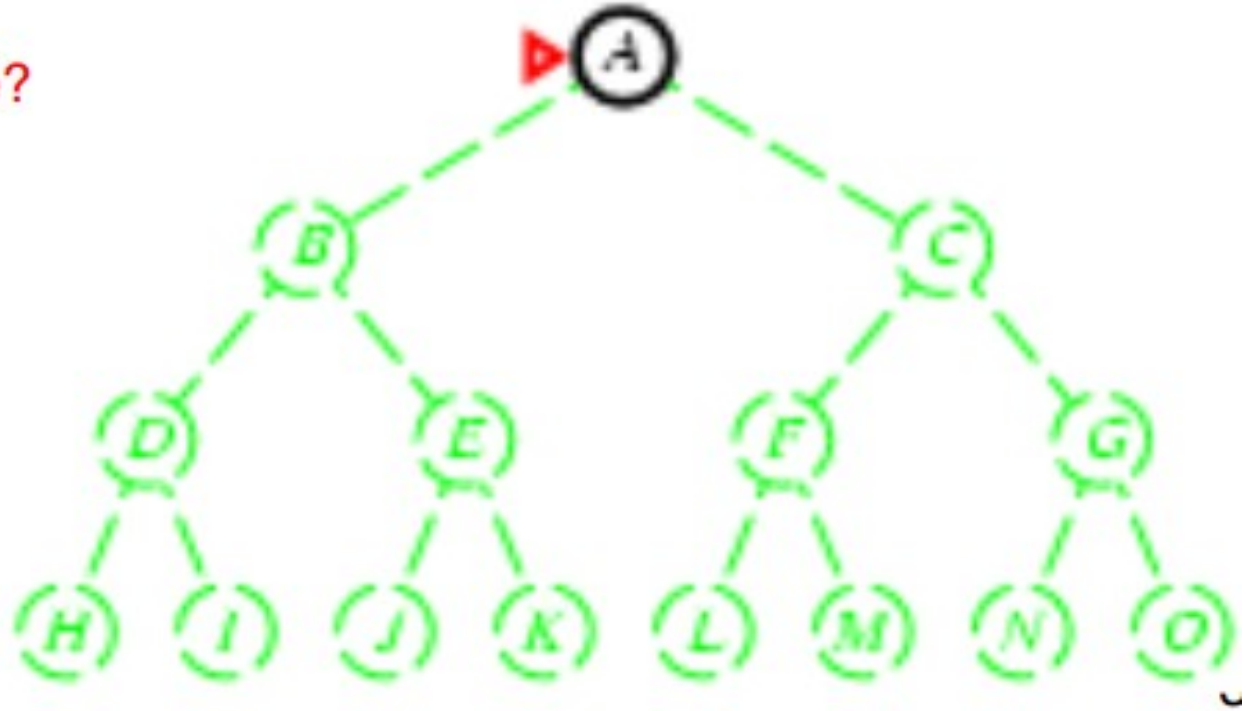
- ✓ Process: Use stack to keep track of nodes. (LIFO)
  - Put the start node on the stack
  - While stack is not empty
    - Pop the stack
    - If the top of stack is the goal, stop
    - Other wiser push the nodes connected to the top of the stack on the stack (provided they are not already on the stack)

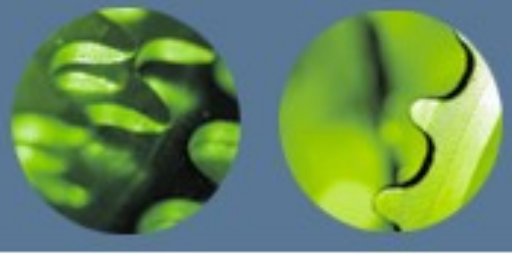


# Depth-First Search (DFS)

- ✓ Find the path from A to M

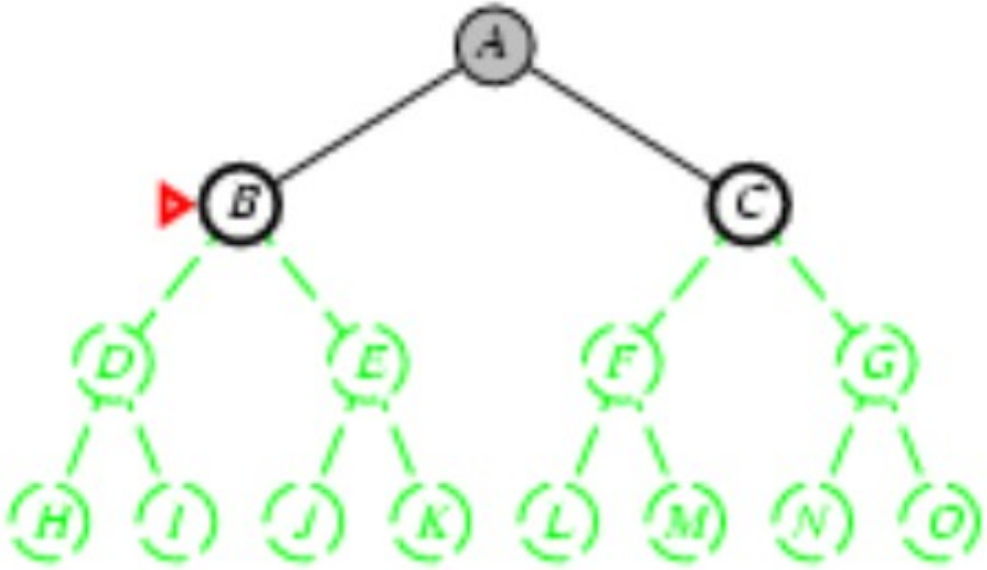
Is A a goal state?

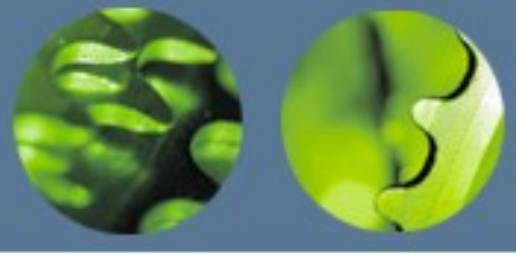




# Depth-First Search (DFS)

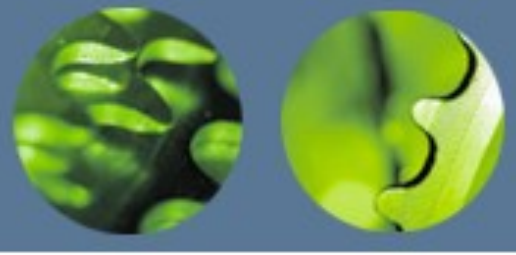
- ✓ Stack = [B, C]
- ✓ Is B a goal state?





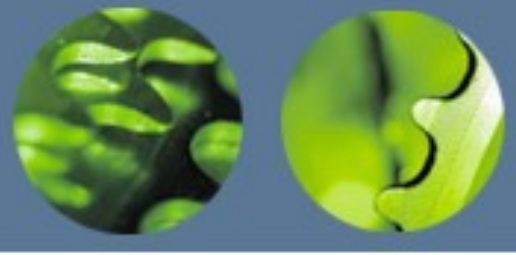
# Depth-First Search (DFS)

- ✓ Stack = [D, E, C]
- ✓ Is D a goal state?



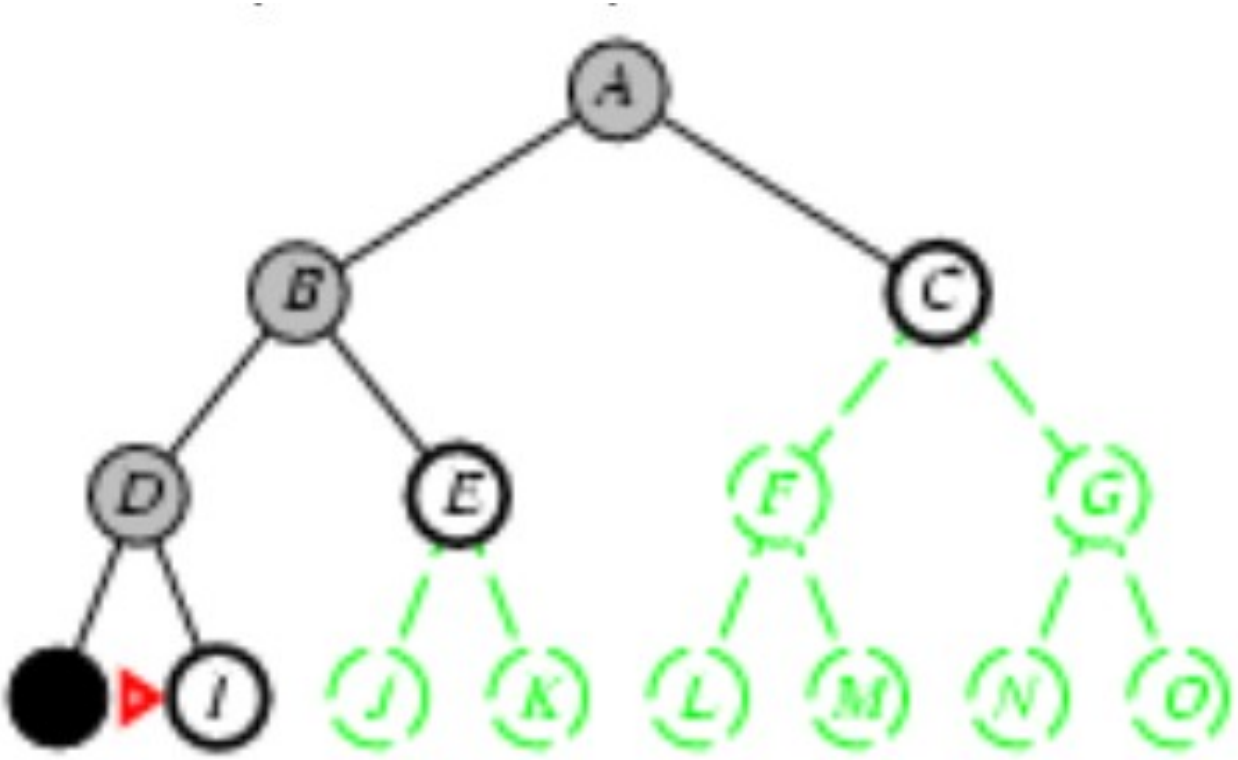
# Depth-First Search (DFS)

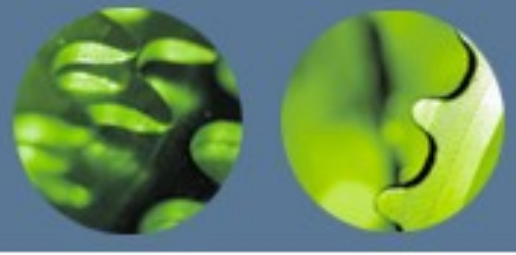
- ✓ Stack = [H, I, E, C]
- ✓ Is H a goal state?



# Depth-First Search (DFS)

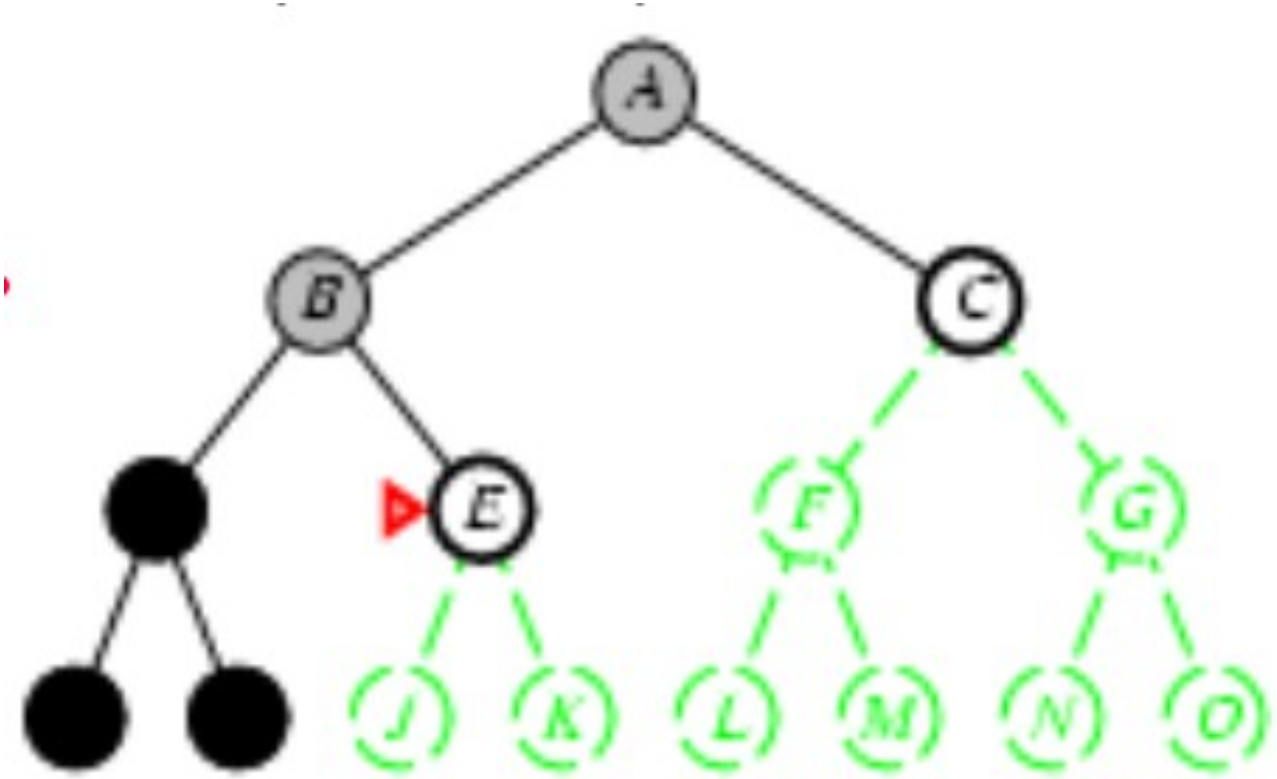
- ✓ Stack = [I, E, C]
- ✓ Is I a goal state?

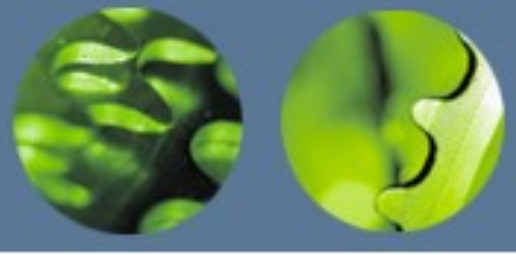




# Depth-First Search (DFS)

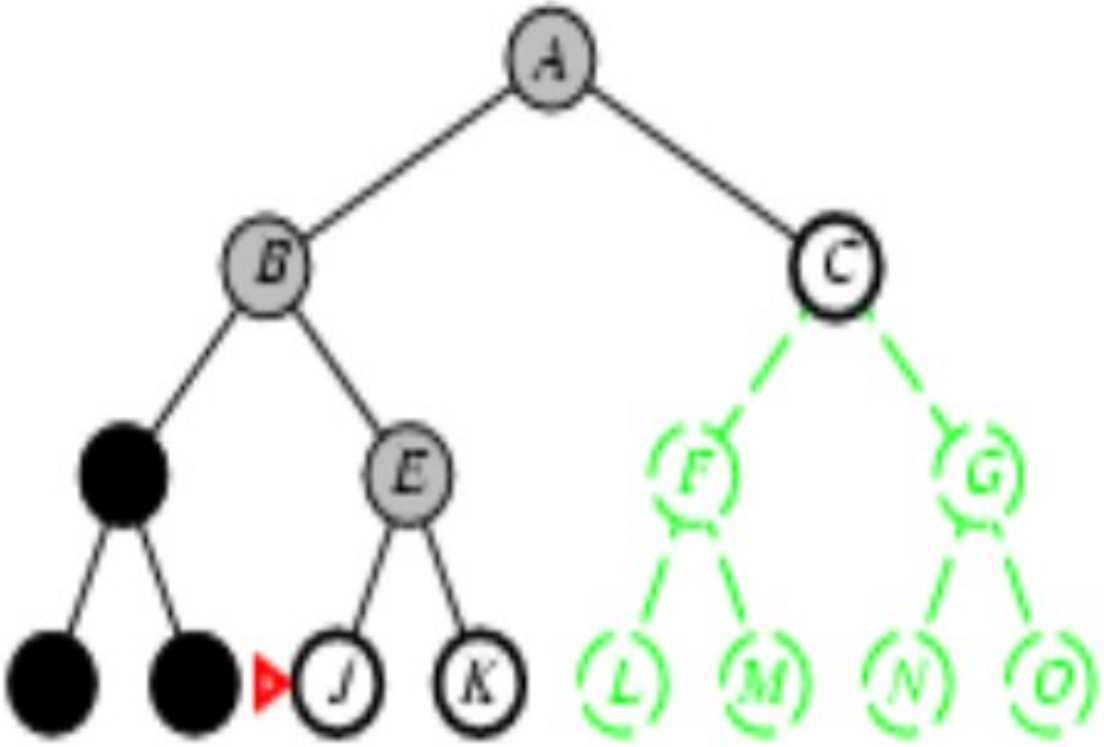
- ✓ Stack = [E, C]
- ✓ Is E a goal state?

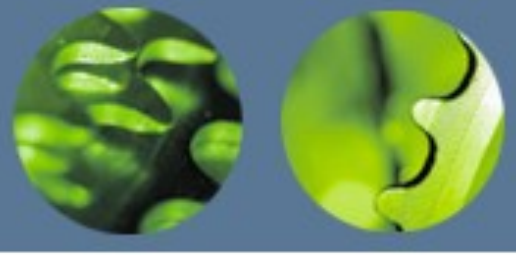




# Depth-First Search (DFS)

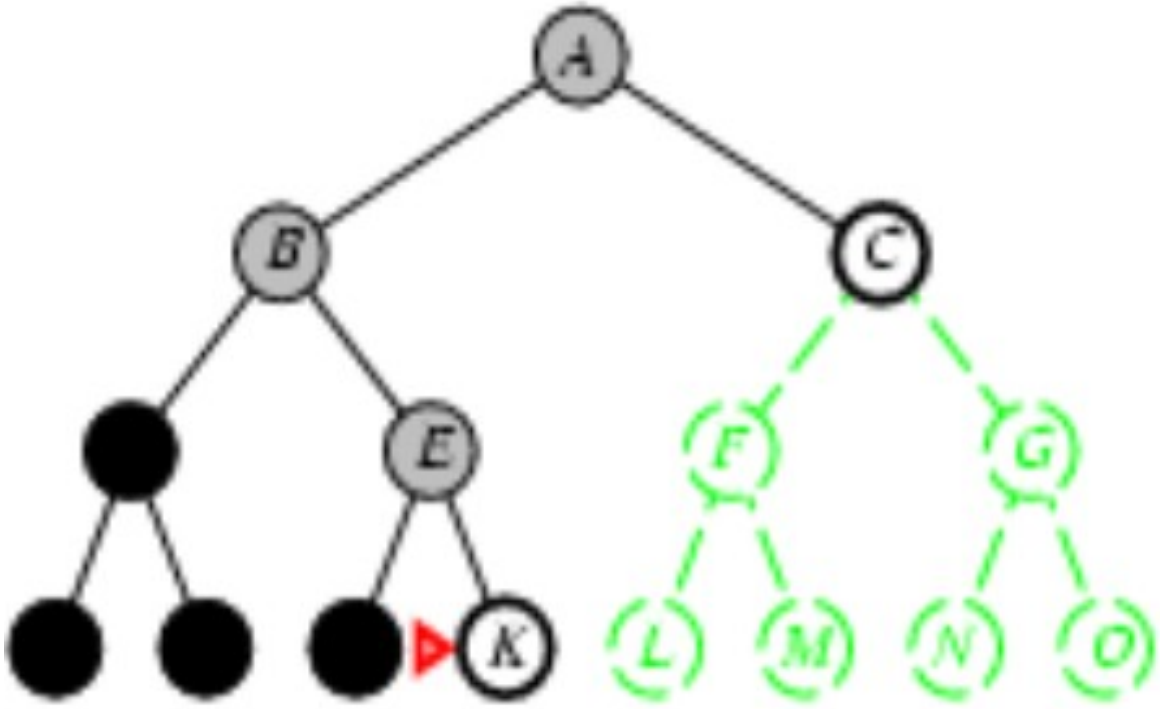
- ✓ Stack = [J, K, C]
- ✓ Is J a goal state?

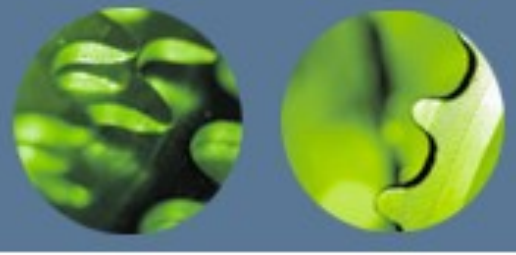




# Depth-First Search (DFS)

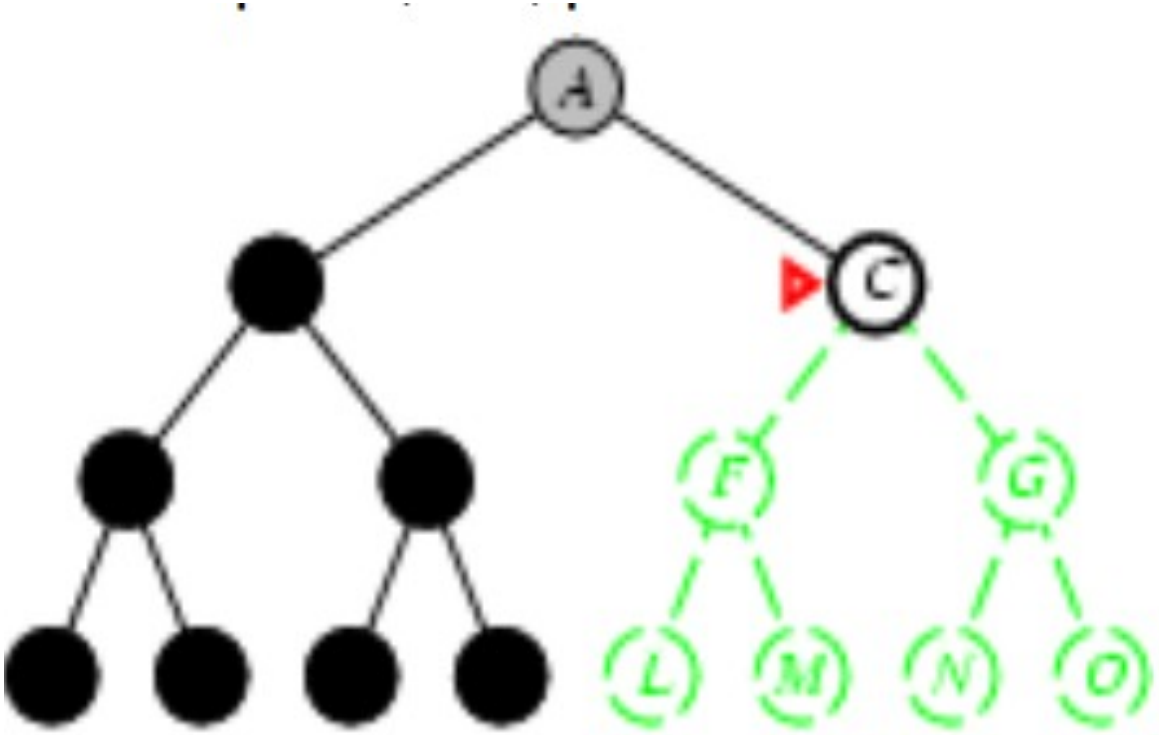
- ✓ Stack = [K, C]
- ✓ Is K a goal state?

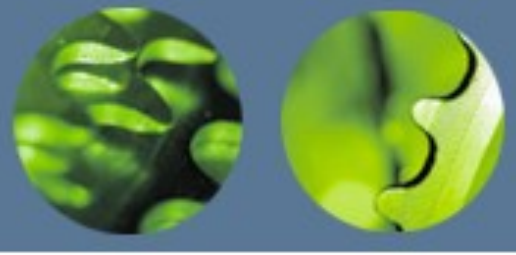




# Depth-First Search (DFS)

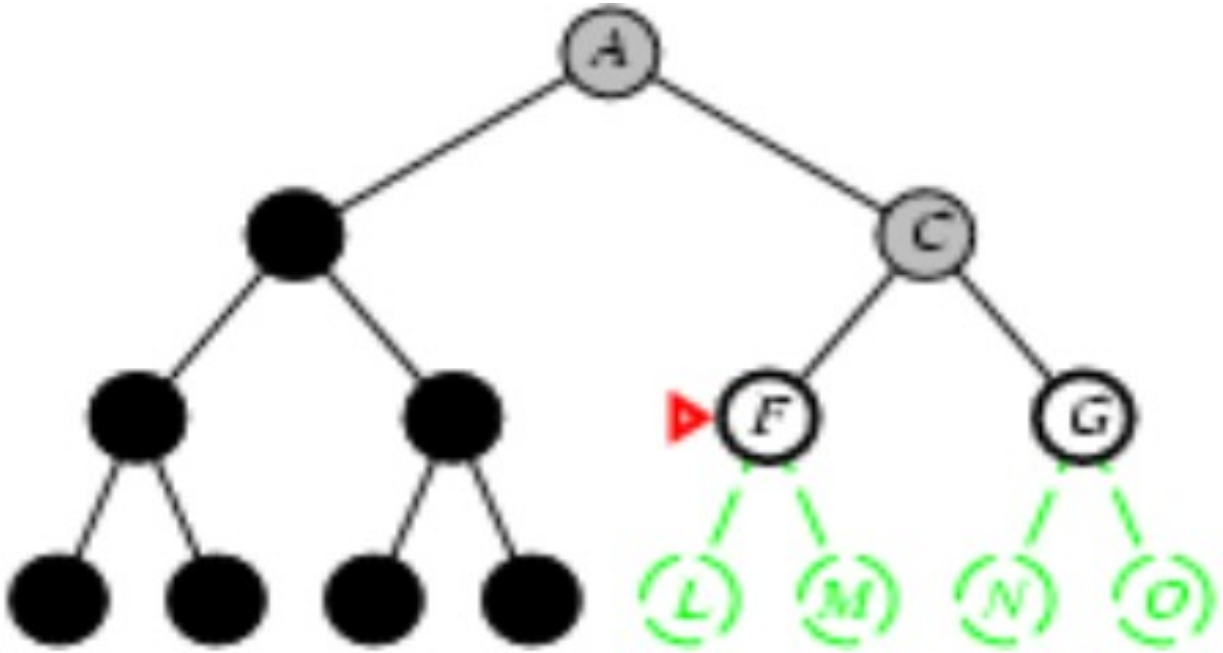
- ✓ Stack = [C]
- ✓ Is C a goal state?

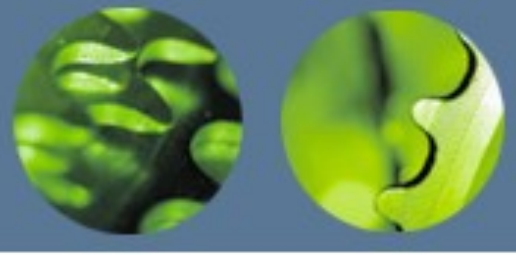




# Depth-First Search (DFS)

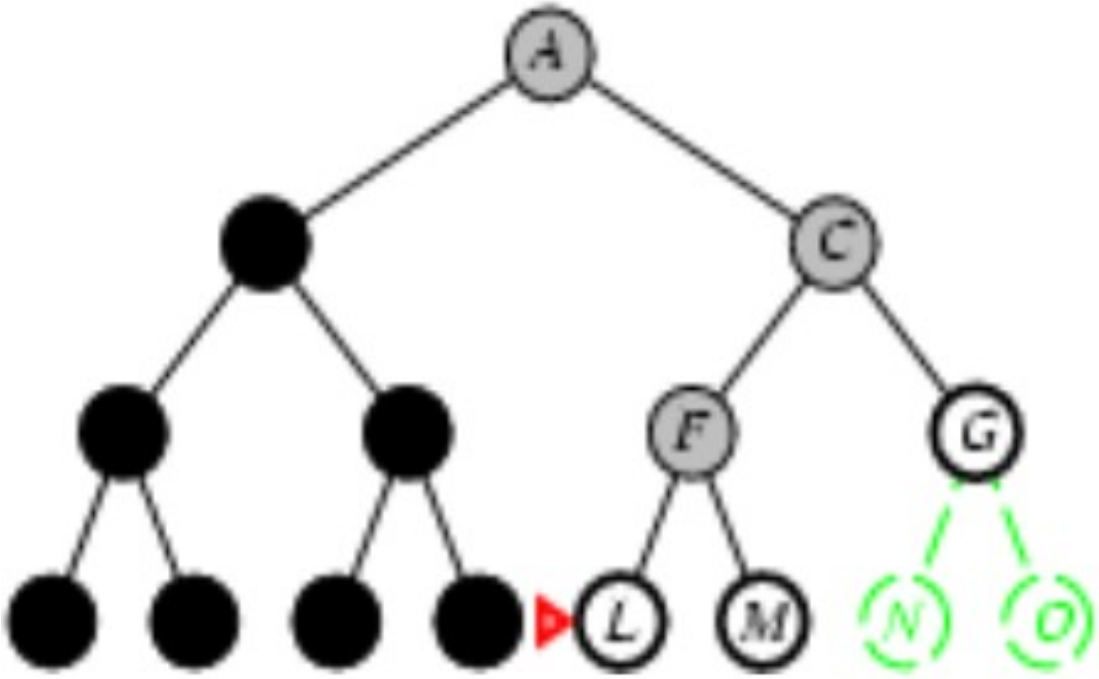
- ✓ Stack = [F, G]
- ✓ Is F a goal state?

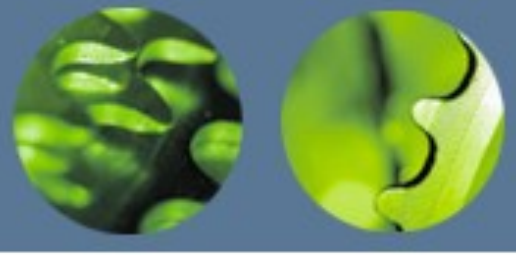




# Depth-First Search (DFS)

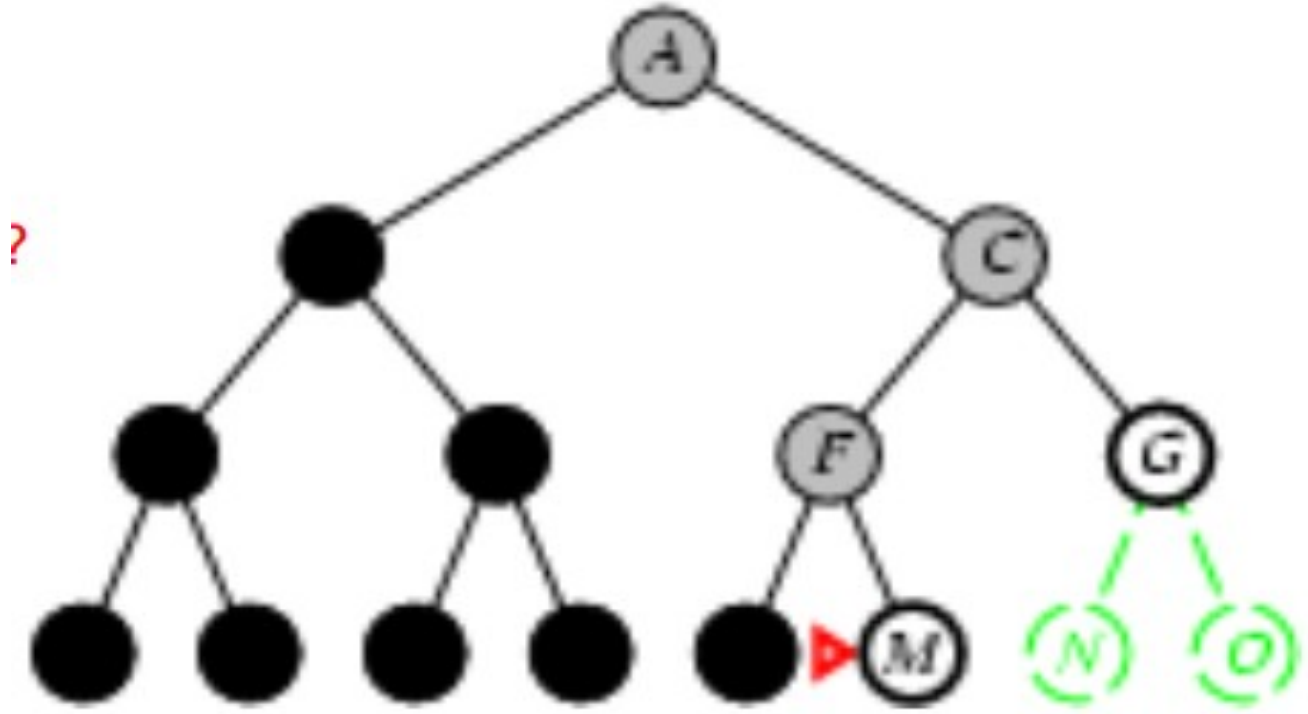
- ✓ Stack = [L, M, G]
- ✓ Is L a goal state?

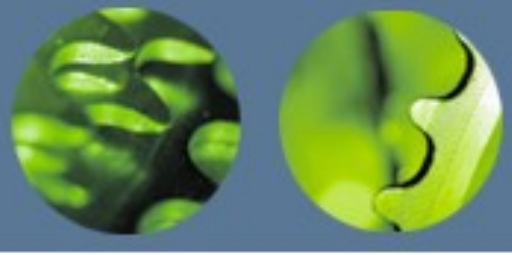




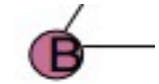
# Depth-First Search (DFS)

- ✓ Stack = [M, G]
- ✓ Is M a goal state?



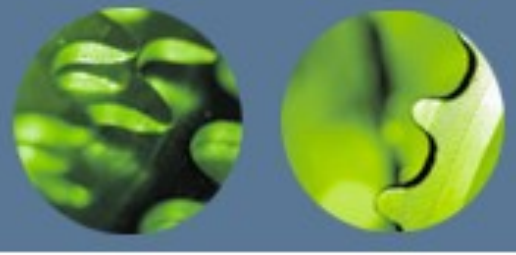


# Properties of DFS




- Complete? No: fails in infinite-depth spaces

  Can modify to avoid repeated states along path
- Time? $O(b^m)$ with m=maximum depth
- terrible if *m* is much larger than *d*
  - but if solutions are dense, may be much faster than breadth-first
- Space? $O(bm)$, i.e., linear space! (we only need to remember a single path + expanded unexplored nodes)
- Optimal? No (It may find a non-optimal goal first)



# Modifications on DFS

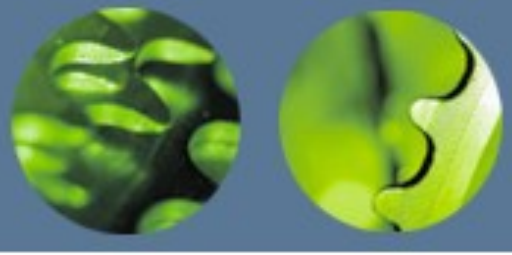
- To avoid the infinite depth problem of DFS, we can decide to only search until depth L, i.e. we don't expand beyond depth L.

  ⇒ Depth-Limited Search

- What of solution is deeper than L? -->Increase L iteratively.

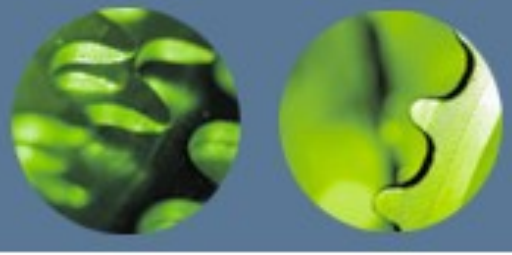
  ⇒ Iterative Deepening Search

- As we shall see: this inherits the memory advantage of Depth-First search, and is better in terms of time complexity than Breadth first search.



# Depth–Limited Search

- ✓ Breadth first has computational, especially, space problems.
- ✓ Depth first can run off down a very long (or infinite) path.
- ✓ Solution may be not be optimal.
- ✓ Hence, Depth limited search.
  - Perform depth first search but only to a per-specified depth limit L.
  - No node on a path that is more than L steps from the initial state is placed on the Frontier.
  - We “truncate” the search by looking only at paths of length L or less.
- ✓ Now infinite length paths are not a problem.
- ✓ But will only find a solution if a solution of length ≤ L exists.

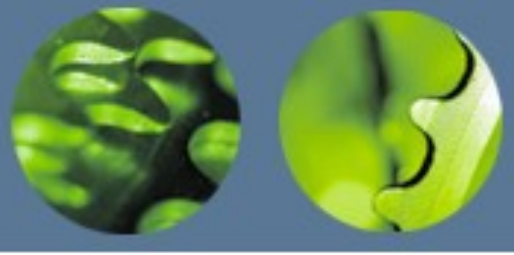


# Depth–Limited Search

## ✓ Properties of Depth–Limited Search

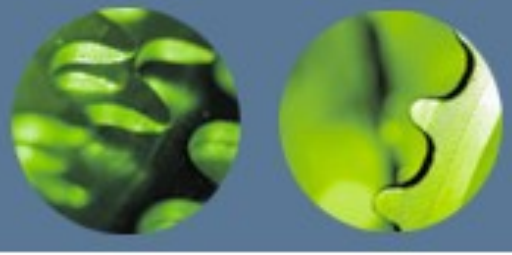
- ***Completeness***: Incomplete as solution may be beyond specified depth level.
- ***Optimality***: not optimal
- ***Space complexity***: b as branching factor and l as tree depth level,

  Space complexity = **O(b.l)**
- ***Time Complexity***: $\mathbf{O(b^l)}$



# Iterative Deepening DFS

- ✓ Takes the idea of depth limited search one step further.
- ✓ Starting at depth limit L = 0, we iteratively increase the depth limit, performing a depth limited search for each depth limit.
- ✓ Stop if no solution is found, or if the depth limited search failed without cutting off any nodes because of the depth limit.
- ✓ Iterative deepening search uses only linear space and not much more time than other uninformed algorithms
- ✓ Search is helpful only if the solution is at given depth level

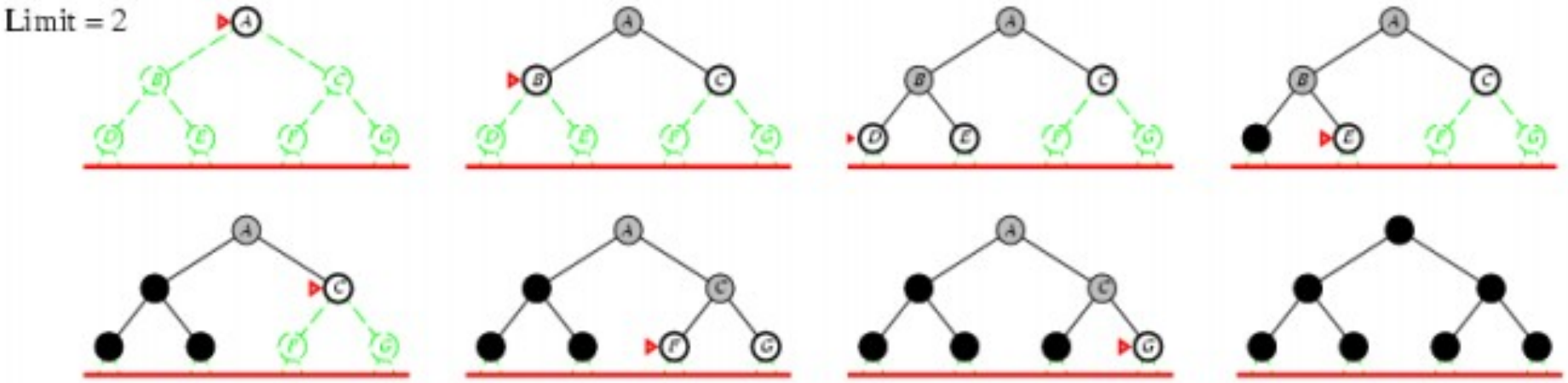


# Iterative Deepening DFS( L = 0)

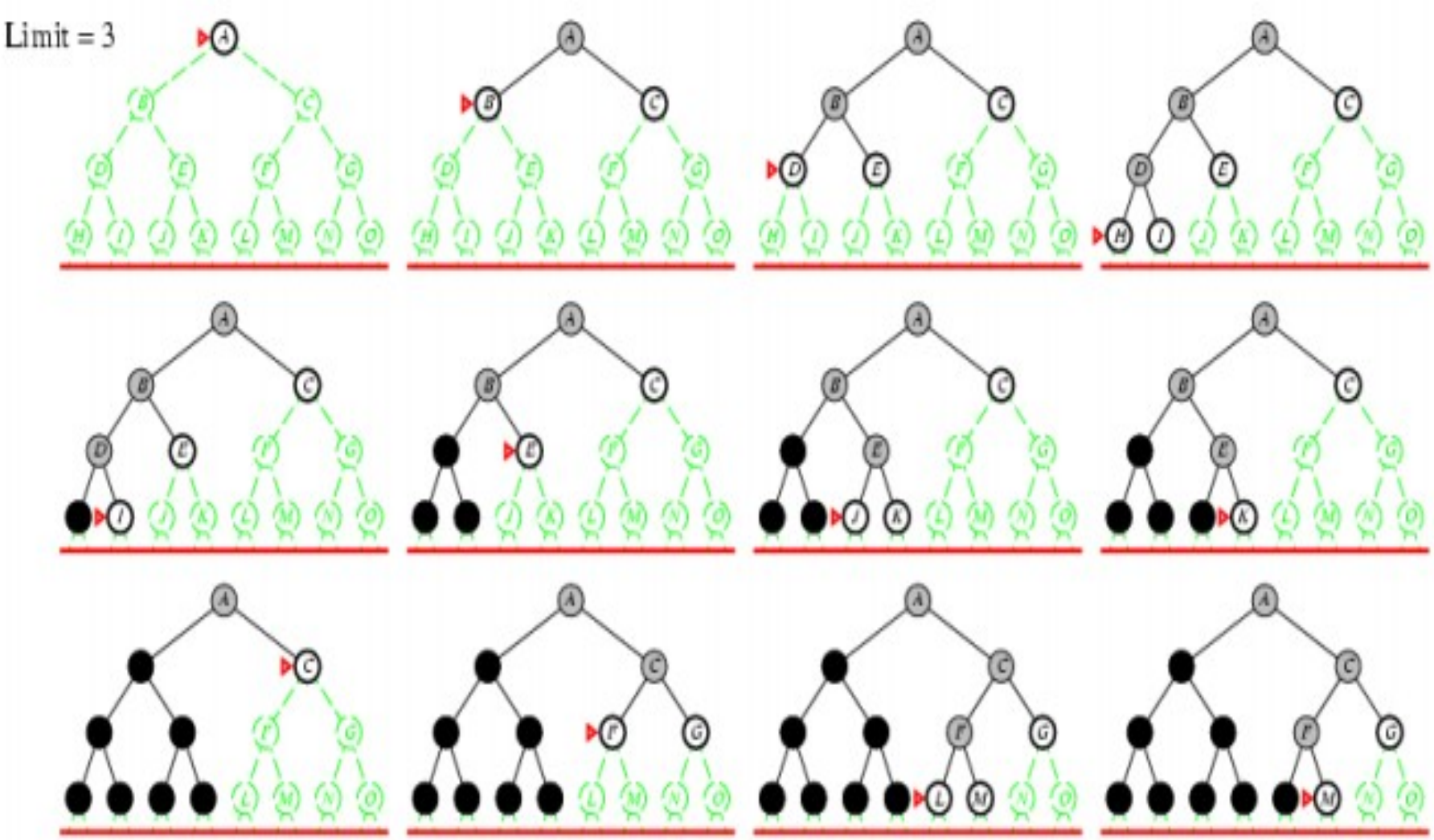
Limit = 0

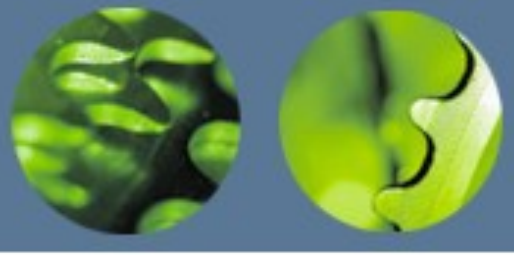

# Iterative Deepening DFS( L = 1)

# Iterative Deepening DFS( L = 2)



# Iterative Deepening DFS( L = 3)





# Iterative Deepening DFS

- Number of nodes generated in a depth-limited search to depth *d* with branching factor *b*:

$$N_{DLS} = b^0 + b^1 + b^2 + \ldots + b^{d-2} + b^{d-1} + b^d$$

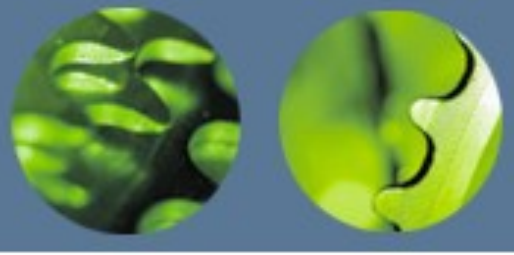
- Number of nodes generated in an iterative deepening search to depth *d* with branching factor *b*:

$$N_{IDS} = (d+1)b^0 + d\,b^1 + (d-1)b^2 + \ldots + 3b^{d-2} + 2b^{d-1} + 1b^d =$$

$O(b^d) \rightarrow O(b^{d+1})$

BFS

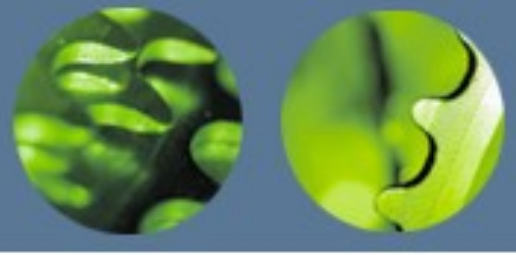
- For *b = 10*, *d = 5*,
  - $N_{DLS} = 1 + 10 + 100 + 1{,}000 + 10{,}000 + 100{,}000 = 111{,}111$
  - $N_{IDS} = 6 + 50 + 400 + 3{,}000 + 20{,}000 + 100{,}000 = 123{,}450$
  - $N_{BFS} = \ldots\ldots = 1{,}111{,}100$



# Iterative Deepening DFS

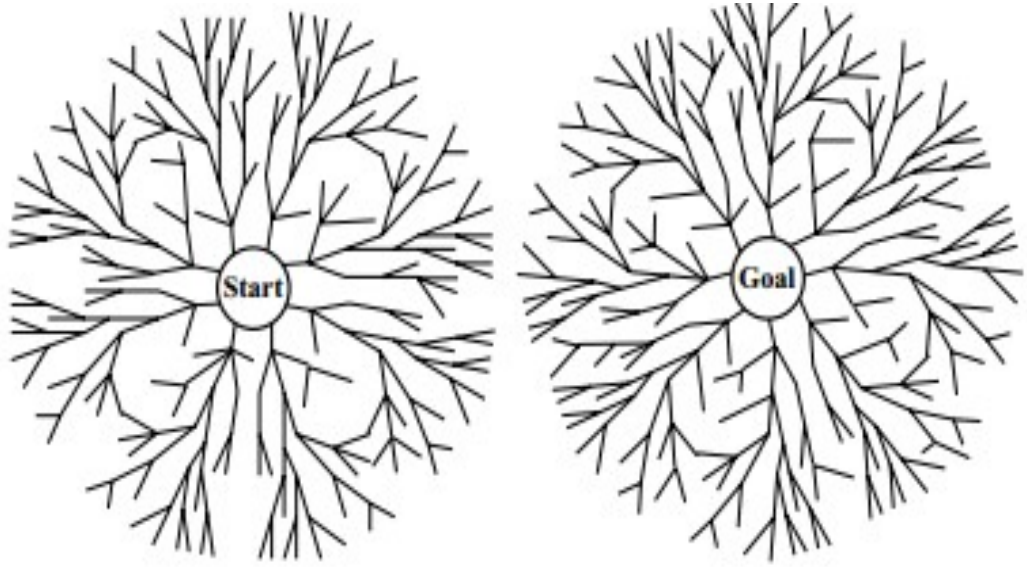
## Properties of Iterative Deepening DFS

- Complete? Yes
- Time? $O(b^d)$
- Space? $O(bd)$
- Optimal? Yes, if step cost = 1 or increasing function of depth.

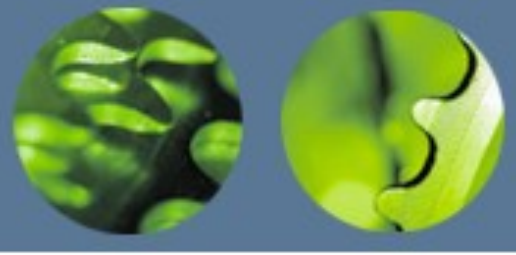


# Bidirectional Search

- ✓ The idea behind bidirectional search is to run two simultaneous searches—one forward from the initial state and the other backward from the goal—hoping that the two searches meet in the middle.



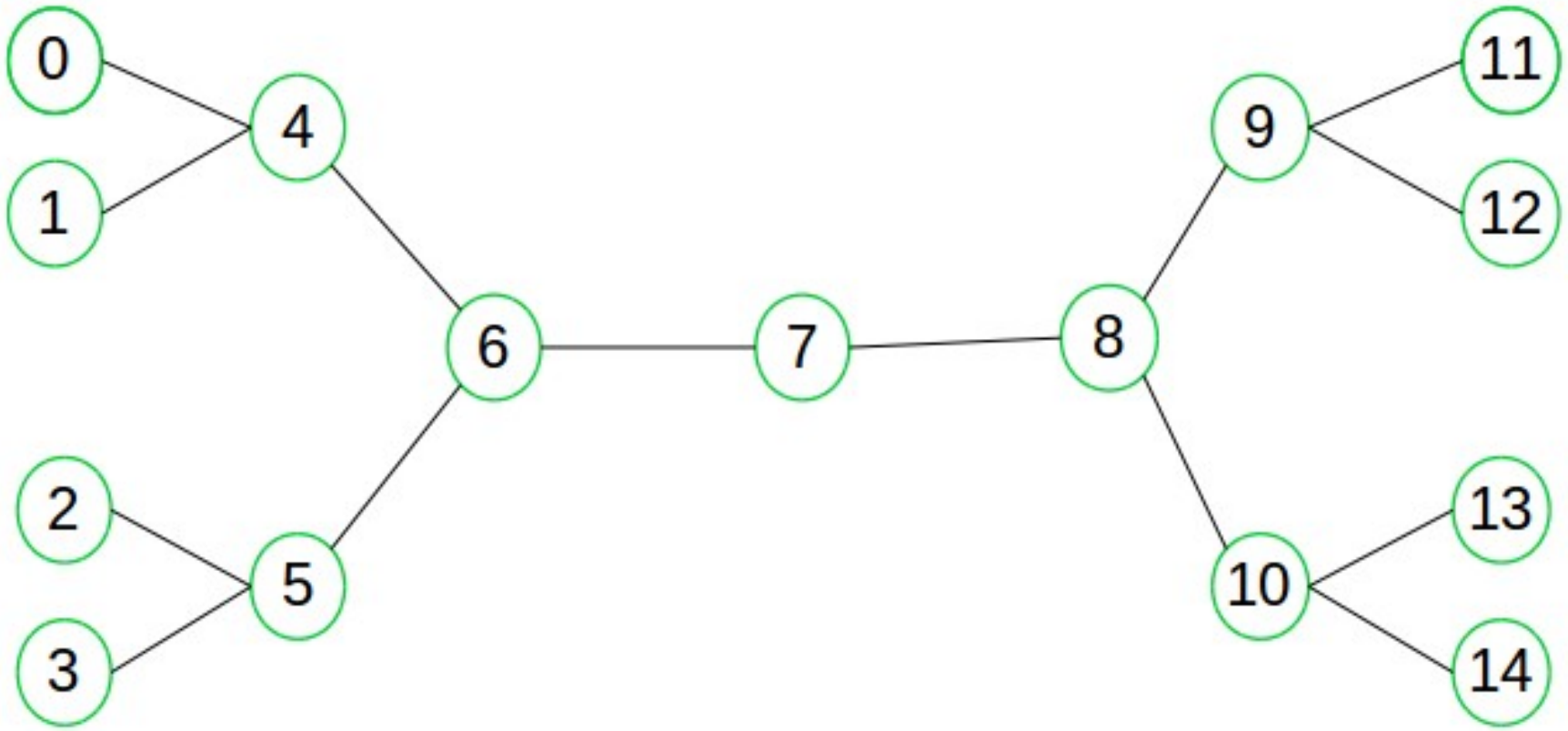

Time complexity: $O(b^{d/2})$. Space complexity: $O(b^{d/2})$

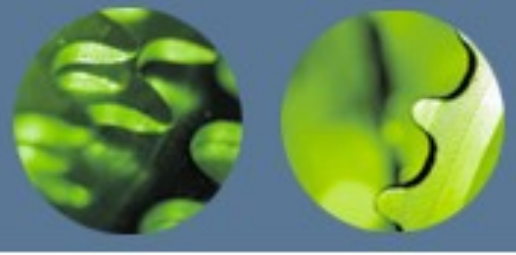


# Bidirectional Search

- ✓ Idea
  - simultaneously search forward from S and backwards from G
  - stop when both “meet in the middle”
  - need to keep track of the intersection of 2 open sets of nodes
- ✓ What does searching backwards from G mean
  - need a way to specify the predecessors of G
    - this can be difficult,
    - e.g., predecessors of checkmate in chess?
  - which to take if there are multiple goal states?
  - where to start if there is only a goal test, no explicit list?

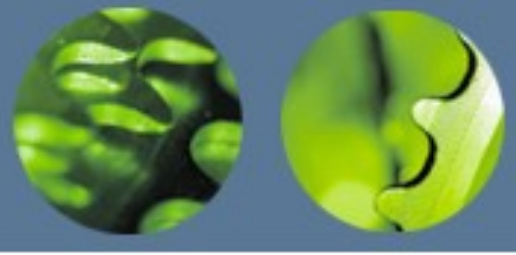
# Bidirectional Search





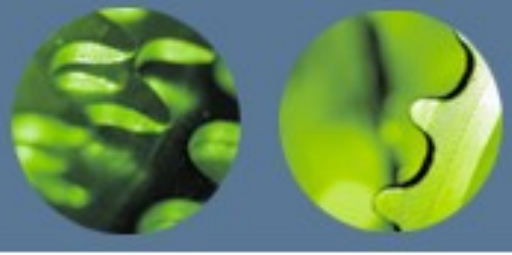
# Bidirectional Search

- ✔ Suppose we want to find if there exists a path from vertex 0 to vertex 14.
- ✔ Here we can execute two searches, one from vertex 0 and other from vertex 14.
- ✔ When both forward and backward search meet at vertex 7, we know that we have found a path from node 0 to 14 and search can be terminated now.
- ✔ We can clearly see that we have successfully avoided unnecessary exploration.

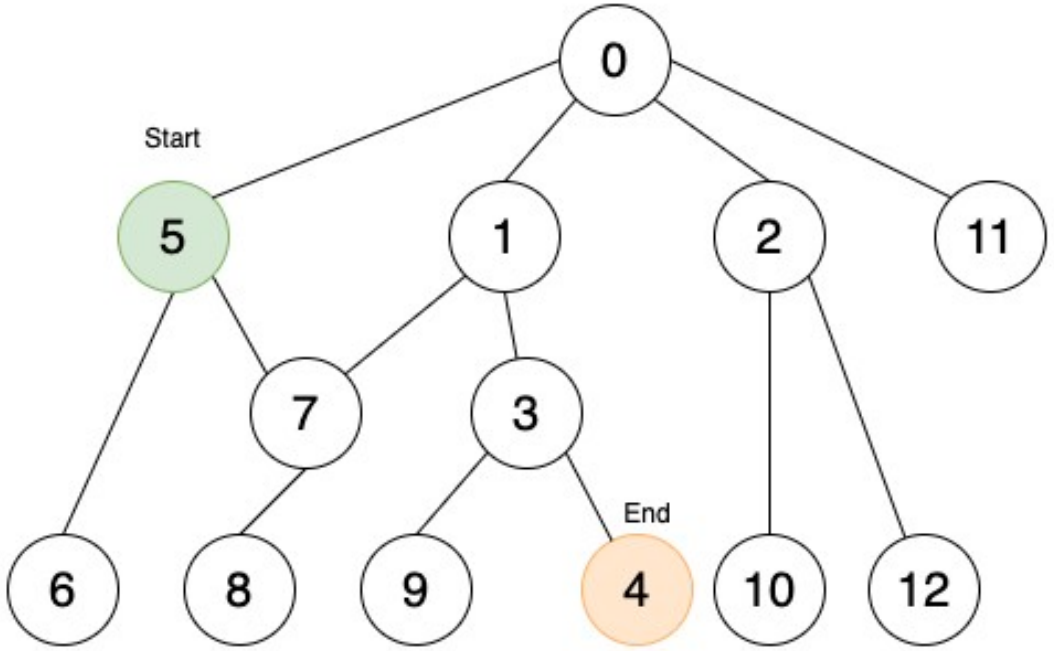


# Bidirectional Search

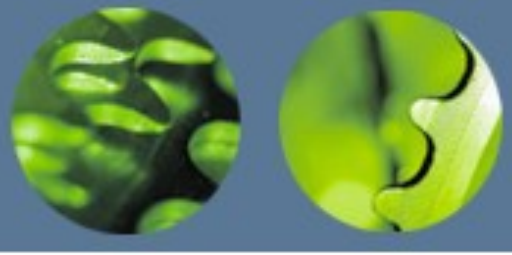
- ✓ Alternate searching from the start state toward the goal and from the goal state toward the start.
- ✓ Stop when the frontiers intersect.
- ✓ Works well only when there are unique start and goal states.
- ✓ Requires the ability to generate “predecessor” states.
- ✓ Can (sometimes) lead to finding a solution more quickly.



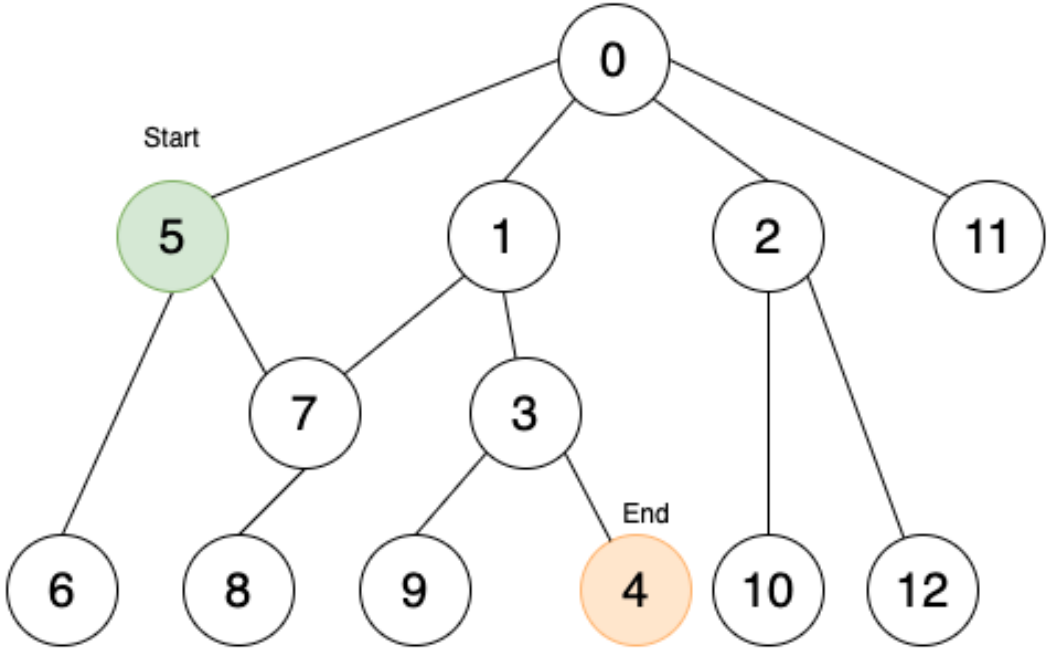
# Bidirectional Search




Aim: To find the shortest path from 5 to 4 using bidirectional search.

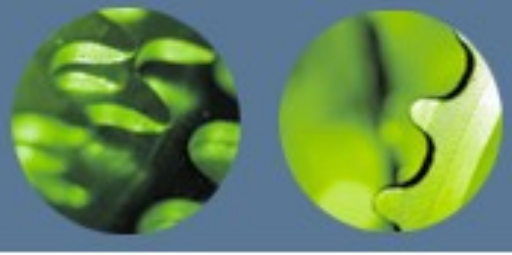


# Bidirectional Search

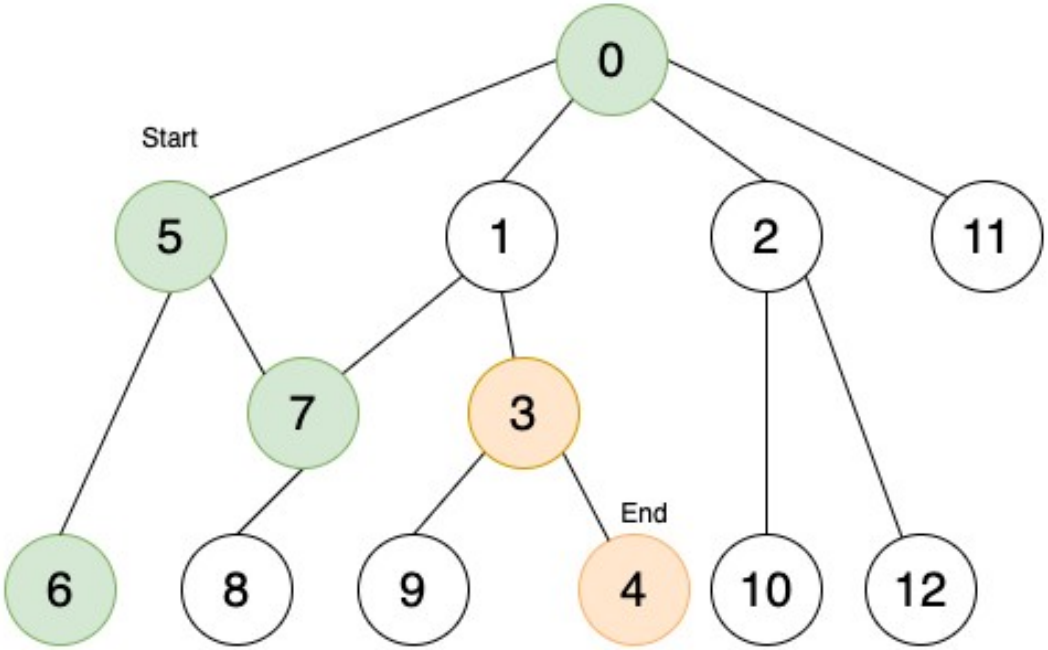



**Do BFS from both directions.**

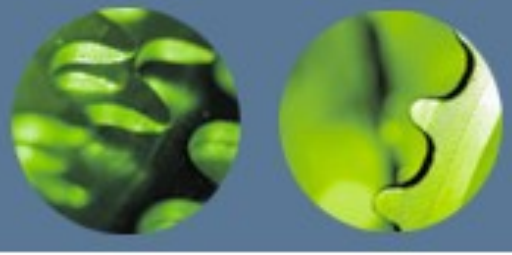
1)Start moving forward from start node (Green) and backwards from end node (Orange).



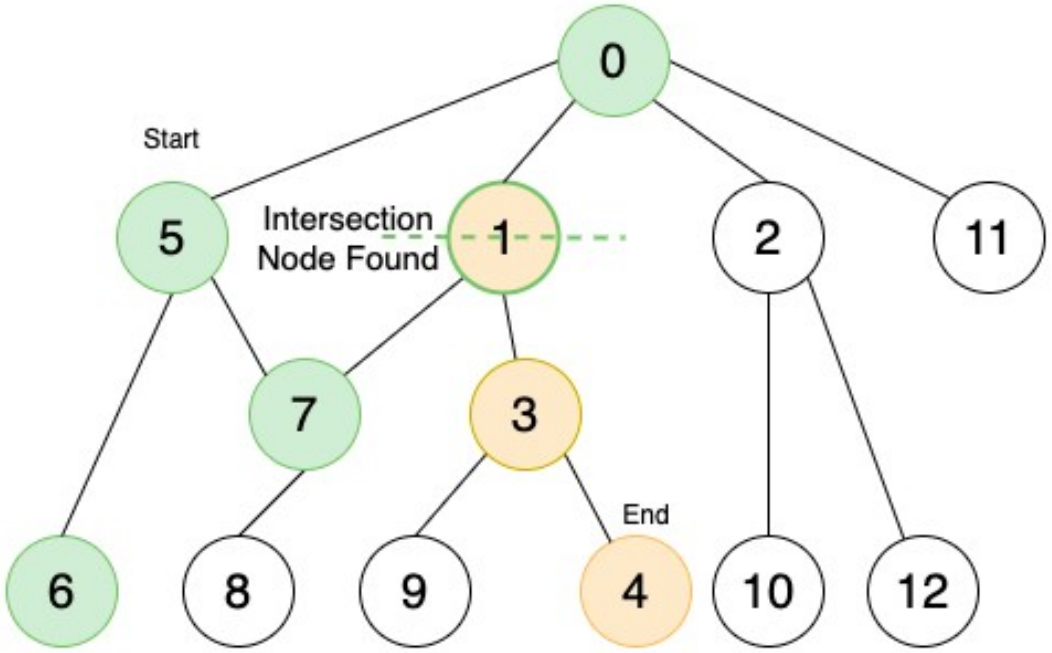
# Bidirectional Search



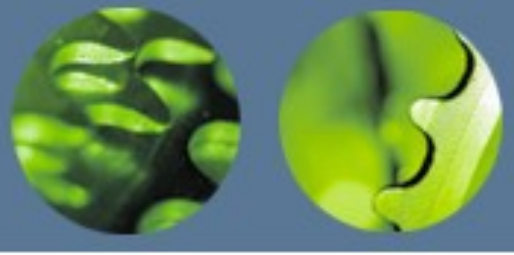

2) Similar to BFS, at every point explore the next level of nodes till you find an intersecting node.



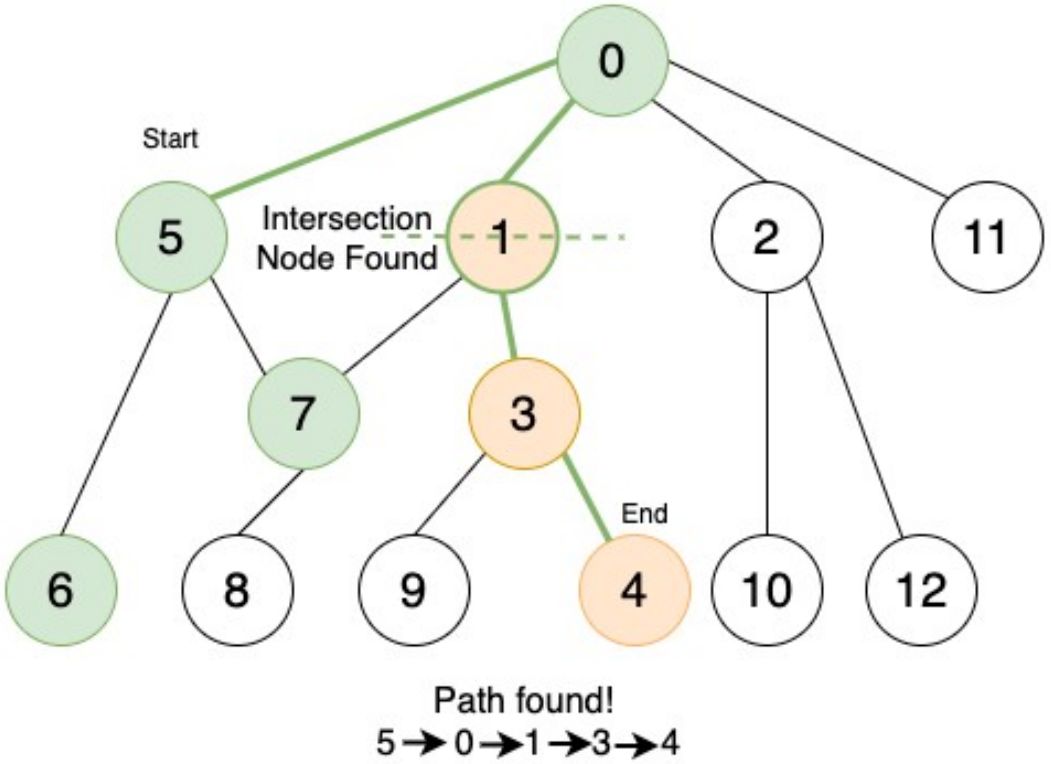
# Bidirectional Search



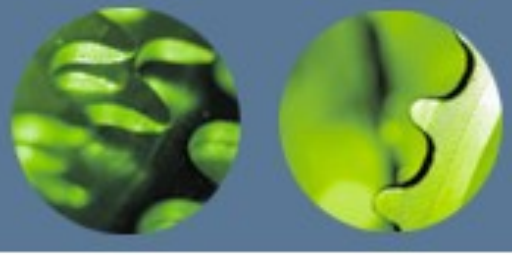

3) Stop on finding the intersecting node.



# Bidirectional Search

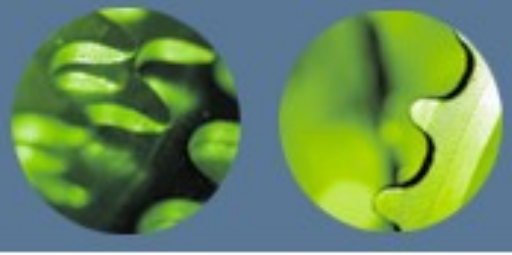



4) Trace back to find the path



# Why Bidirectional Search?

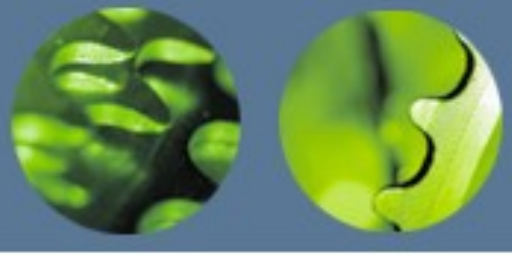
- ✔ Because in many cases it is faster, it dramatically reduce the amount of required exploration.
- ✔ Suppose if branching factor of tree is b and distance of goal vertex from source is d, then the normal BFS/DFS searching complexity would be O(b^d).
- ✔ On the other hand, if we execute two search operation then the complexity would be O(b^{d/2}) for each search and total complexity would be O(b^{d/2}+b^{d/2}) which is far less than O(b^d).



# When to use bidirectional approach?

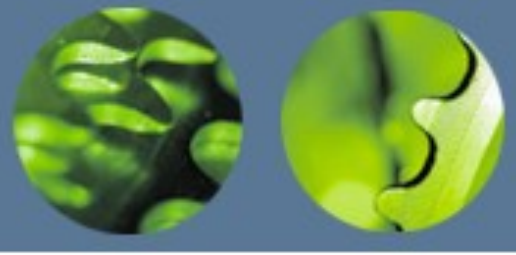
We can consider bidirectional approach when-

- Both initial and goal states are unique and completely defined.
- The branching factor is exactly the same in both directions.



# Performance Measures

- Completeness : Bidirectional search is complete if BFS is used in both searches.
- Optimality : It is optimal if BFS is used for search and paths have uniform cost.
- Time and Space Complexity : Time and space complexity is $O(b^{d/2})$



# Comparing Search Strategies

| Criterion | Breadth-First | Uniform-Cost | Depth-First | Depth-Limited | Iterative Deepening | Bidirectional (if applicable) |
|---|---|---|---|---|---|---|
| Complete? | Yes[a] | Yes[a,b] | No | No | Yes[a] | Yes[a,d] |
| Time | $O(b^d)$ | $O(b^{1+\lfloor C^*/\epsilon \rfloor})$ | $O(b^m)$ | $O(b^\ell)$ | $O(b^d)$ | $O(b^{d/2})$ |
| Space | $O(b^d)$ | $O(b^{1+\lfloor C^*/\epsilon \rfloor})$ | $O(bm)$ | $O(b\ell)$ | $O(bd)$ | $O(b^{d/2})$ |
| Optimal? | Yes[c] | Yes | No | No | Yes[c] | Yes[c,d] |

**Figure 3.21** Evaluation of tree-search strategies. $b$ is the branching factor; $d$ is the depth of the shallowest solution; $m$ is the maximum depth of the search tree; $l$ is the depth limit. Superscript caveats are as follows: [a] complete if $b$ is finite; [b] complete if step costs $\geq \epsilon$ for positive $\epsilon$; [c] optimal if step costs are all identical; [d] if both directions use breadth-first search.